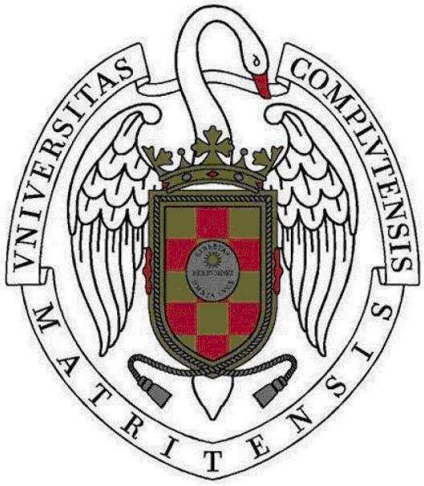

DESARROLLO DE UN COMPILADOR DE UNA REPRESENTACIÓN BASADA EN REGLAS A CÓDIGO DE BYTES DE JAVA

Autores:

Luis Celada Trigo
Carlos Gil Jiménez

Profesora directora:

Elvira María Albert Albiol

Proyecto de Sistemas Informáticos
Facultad de Informática
Universidad Complutense de Madrid

Índice

Autorización a la difusión

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales, y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Luis Celada Trigo

Carlos Gil Jiménez

Agradecimientos

En primer lugar queremos agradecer la ayuda prestada a nuestra profesora directora, Elvira María Albert Albiol.

Además, agradecer el apoyo prestado a nuestras familias y amigos durante todo el desarrollo del proyecto.

Por último, también agradecer a todo aquel que quiera leer este proyecto, con la intención de que le sirva de ayuda.

Resumen del proyecto

El objetivo de este proyecto es la realización de un compilador cuyo lenguaje fuente es una representación intermedia basada en reglas, y su lenguaje objeto es un lenguaje próximo al bytecode de Java, sin llegar a serlo, es el lenguaje de entrada de la aplicación ***Jasmin***, que genera como salida un fichero ***.class***.
Nuestro proyecto consta de tres pasos. El primero, es la decompilación de programas en bytecode a esa representación intermedia. La motivación de hacer esa traducción es hacer análisis de consumo de recursos y terminación de programas en código de byte de Java (que posiblemente utilizan las librerías de Java. Es de lo que se encarga el sistema ***COSTA***.
El segundo paso es, con la esperanza en un futuro de haber hecho optimizaciones de código de esos programas, trabajando con las reglas, volver a compilar ese programa al bytecode de Java, en el que podemos distinguir otros dos pasos. El primero es la generación de un archivo ***.j***, el que acepta la aplicación ***Jasmin***, cuya función es ensamblar ese archivo generando un .***class***, añadiendo las referencias simbólicas, que es el lo que consiste el tercer paso.

The aim of this project is the development of a compiler whose source language is an intermediate rule-based representation, and its object code is a language close to Java bytecode, homely it is the language of the ***Jasmin*** application.
Our project consists of three steps. The first step is the decompilation of Java bytecode to that intermediate representation. The motivation to make that translation is to perform resource bound and termination analyses for programs that possibly use Java libraries. This is the function of the ***COSTA*** system.
The second step is, with the idea of making code optimizations to those algorithms in the future working with the rules, to translate that program again, from the rules to Java bytecode, distinguishing two other steps:
The first one is the ***J*** File generation, which is the input of the Jasmin application, whose goal is to assemble that file making a ***CLASS*** File, adding the symbolic references, being this the third step.

Introducción

La motivación de este proyecto es la de realizar análisis de distintas categorías sobre programas en bytecode con el objetivo final de hacer optimizaciones de código. Si en un futuro no se tuviera como objetivo la realización de esas optimizaciones no tendría sentido nuestra labor, ya que bastaría con hacer los análisis sobre el conjunto de reglas, y no hacer nada más, pues ya existirían los programas en bytecode originales.

Nuestra labor es muy funcional, cuando en un futuro esas optimizaciones sean una realidad, la implementación de este proyecto, devolverá el código a su forma original: el bytecode, preservando su flujo de control.

En la figura de abajo, se puede ver el proceso de transformaciones a los que se ven sometidos estos programas. Nuestra parte abarca la compilación desde la representación intermedia al fichero .class; los pasos anteriores ya están implementados dentro del sistema ***COSTA***.

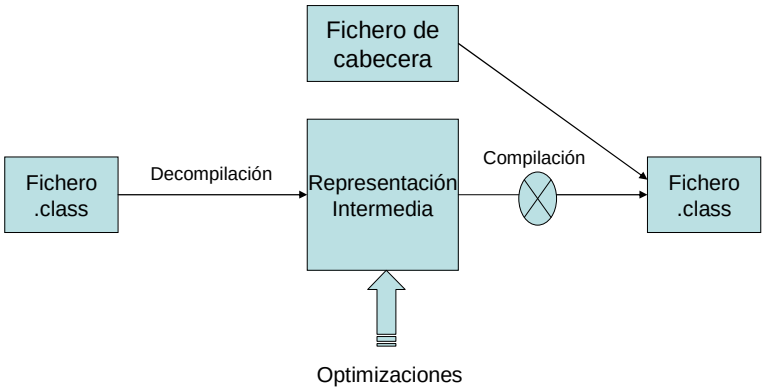

Figura 1. Estructura general del sistema

Capítulo 1

La Máquina Virtual de Java

1.1 Definición de un procesador del lenguaje

Como ya hemos visto previamente, la razón principal de este proyecto es la realización de un compilador de la representación intermedia que utiliza ***COSTA*** al bytecode de Java. Para entender realmente el proyecto es necesario el repaso de algunos conceptos básicos:

Un traductor es un programa que realiza una conversión de un programa en un lenguaje fuente al mismo programa en un lenguaje objeto. Existen varios tipos de traductores, también llamados procesadores del lenguaje. Los que operan a más alto nivel, es decir, hacen la transformación de un lenguaje fuente a un lenguaje objeto, que suele ser un fichero binario con una representación establecida según términos básicos, respetando su semántica operacional, cercano al código máquina en cierto sentido, ya que éste último siempre dependerá del juego de instrucciones de la arquitectura del computador. Otro tipo de traductor son los ensambladores, que realizan la traducción de un lenguaje objeto directamente al lenguaje máquina de la arquitectura en si. Si bien un compilador de un lenguaje de programación, puede ser común en cada equipo, ya que el código objeto siempre es el mismo, un ensamblador es diferente según la estructura interna del mismo.

En contraposición a los compiladores, existen otro tipo de traductores, los intérpretes. Éstos lo que hacen es ejecutar directamente el código “línea a línea”, sin necesidad de traducir el programa a un código intermedio. En la actualidad el ejemplo más claro de intérprete es el parser de los navegadores, que ejecutan el código de las diferentes tecnologías Web a medida que se va solicitando. Otros lenguajes de programación también son interpretados, y son en su mayor parte, los lenguajes basados en scripts.

1.2 Definición de una máquina virtual

Según las definiciones anteriores, si nosotros desarrollamos un programa en cualquier lenguaje de programación compilado, los pasos que realizará, en este caso, el compilador, serán:

1. Primero, habrá que convertir el lenguaje fuente en el lenguaje objeto, para ello, el compilador hace una serie de análisis del programa fuente (generación de la tabla de símbolos, descomposición en tokens por parte del analizador léxico, creación de una estructura intermedia, comprobación de tipos, realización del análisis sintáctico y traducción formal del código, principalmente).

2. Una vez obtenido el código objeto, el siguiente paso es convertirlo a código máquina para que el programa pueda ser ejecutando por el computador en sí, es decir, traducirlo a lenguaje ensamblador con el respectivo conjunto de instrucciones de cada arquitectura.

Éste es el modo de ejecución tradicional de un programa, sin embargo, existe una alternativa que dota a estos sistemas de una gran portabilidad, son las máquinas virtuales.

Una máquina virtual es un programa software capaz de emular a un computador real; esto es, desarrolla las funciones básicas de un computador. De esta manera, nosotros podemos ejecutar un código objeto en una máquina virtual, obteniendo el mismo resultado que si lo ejecutáramos realmente. De esta manera podríamos compilar un programa en cualquier sistema operativo y en cualquier arquitectura, y ejecutarlo en cualquier otra con el mismo resultado. Las máquinas virtuales se pueden usar para probar cualquier programa, incluidos los sistemas operativos, según sean máquinas virtuales de proceso (programas), o de sistema (sistemas operativos). Otra ventaja, es que de esta forma nos acogemos a las abstracciones que proporciona la máquina virtual.

La principal desventaja de las máquinas virtuales es que al ser un programa software, consume un alto porcentaje de los recursos del sistema, ya que son programas muy complejos, que, para emular un sistema real, consumen mucho tiempo de procesador y gran parte de la memoria.

1.3 Plataforma Java

1.3.1 Introducción

El lenguaje de programación de Java fue desarrollado por Sun Microsystems a principios de los años 90. Supuso un gran avance en los lenguajes de programación cuyo paradigma es el orientado a objetos debido a su gestión de memoria, a su procesamiento paralelo o, en lo que nos concierne a nosotros, a la inclusión de su máquina virtual. Las múltiples versiones de la plataforma Java son software libre (exceptuando las librerías, que aún no están consideradas como tal), y están bajo la licencia GNU/GPL. Las versiones se

suelen distribuir en el denominado kit de desarrollo de Java (JDK-J2SE), que incorpora no sólo el compilador (***javac***), sino multitud de aplicaciones, siendo alguna de ellas importante para nuestra tarea en particular, como por ejemplo, el desensamblador de código (***javap***).

1.3.2 COMPILACIÓN DE UN PROGRAMA JAVA

De esta manera, si queremos ejecutar un programa escrito en el lenguaje de programación de Java, primero se compilará con la utilidad ***javac*** (java compiler), bien desde línea de comandos, o bien desde un entorno de desarrollo, en la plataforma que nosotros queramos. La extensión del fichero fuente será ***.java***, y una vez compilado, lo que obtendremos será un ***.class***, del que hablaremos más tarde. Ese fichero binario será el bytecode. El segundo paso es interpretar ese bytecode en la máquina virtual de Java (Java Virtual Machine), que también está integrada en el JDK/J2SE. La máquina virtual ejecutará el programa y producirá la salida, independiente de la plataforma en la que ejecutemos el ***.class***.

La máquina virtual de Java funciona mediante un sistema en tiempo real que, como en casi todas las máquinas virtuales, su idea principal de implementación es una máquina finita de estados.

No todos los compiladores del lenguaje de programación de Java funcionan así. Existen los llamados traductor JIT (Just in Time) que lo que hacen es juntar los 2 pasos en uno solo, es decir, traducen el bytecode de Java a código máquina dependiente de la arquitectura en tiempo de ejecución del programa. Esta idea tiene múltiples desventajas, ya que desechan la idea de la portabilidad de la máquina Virtual de Java, aunque permanezca implícita la portabilidad del bytecode por si mismo.

1.3.3 UTILIDADES

El kit de desarrollo de Java contiene varias aplicaciones relacionadas con el uso del lenguaje de la plataforma Java, las más conocidas son ***javac***, ***jdb***, ***appletviewer***…. Pero para la realización de este proyecto vamos a seleccionar dos.

Java Compiler, o ***javac***, es una utilidad que se proporciona en el software Java Development Kit; más concretamente, es el compilador de Java. Su función es la de traducir un programa escrito en código fuente Java al código de byte. Es una aplicación nativa escrita, a su vez en Java.

Podemos ejecutar esta aplicación o bien desde el shell o línea de comandos, o bien desde un entorno de desarrollo, siempre habiendo ajustado previamente la variable de entorno ***PATH*** del sistema operativo para que éste pueda procesar su ruta. Entornos de desarrollo conocidos son Eclipse, NetBeans (desarrollado por la propia compañía Sun), JCreator, JBuilder….

La aplicación ***javap*** es el desensamblador de código; esto es, hace legible un fichero .***class*** mostrando su contenido, lo que es básico para este proyecto si se pretende hacer traducciones de bytecode.

1.3.4 EL VERIFICADOR DE BYTECODE

Previamente a ejecutarse, la máquina virtual de Java verifica todo bytecode compilado con anterioridad.

La verificación consiste básicamente en tres tipos de revisiones:

- Los saltos siempre deben ser a direcciones válidas
- Los datos siempre se inicializan y siempre se chequea que el tipo de las referencias sea el correcto
- El acceso a los elementos cuyo modificador de acceso es privado es fuertemente controlado

Los dos primeras revisiones son realizadas al cargar el fichero ***.class*** en la máquina virtual de Java, la tercera se realiza a medida que se va ejecutando un determinado flujo de control.

Además, el verificador de bytecode no permite cualquier secuencia de código aunque el programa sea lógicamente válido; por ejemplo, no permite saltos a instrucciones que se encuentren en un método o subrutina distintos de la instrucción que produce el salto. También ofrece protección a las zonas reservadas de la memoria sin tener que recurrir a la unidad de gestión de la memoria (***MMU***) del sistema operativo.

El verificador de bytecode combina todas estas posibilidades, con el recolector de basura de la máquina virtual de Java.

1.3.5 DIAGRAMA

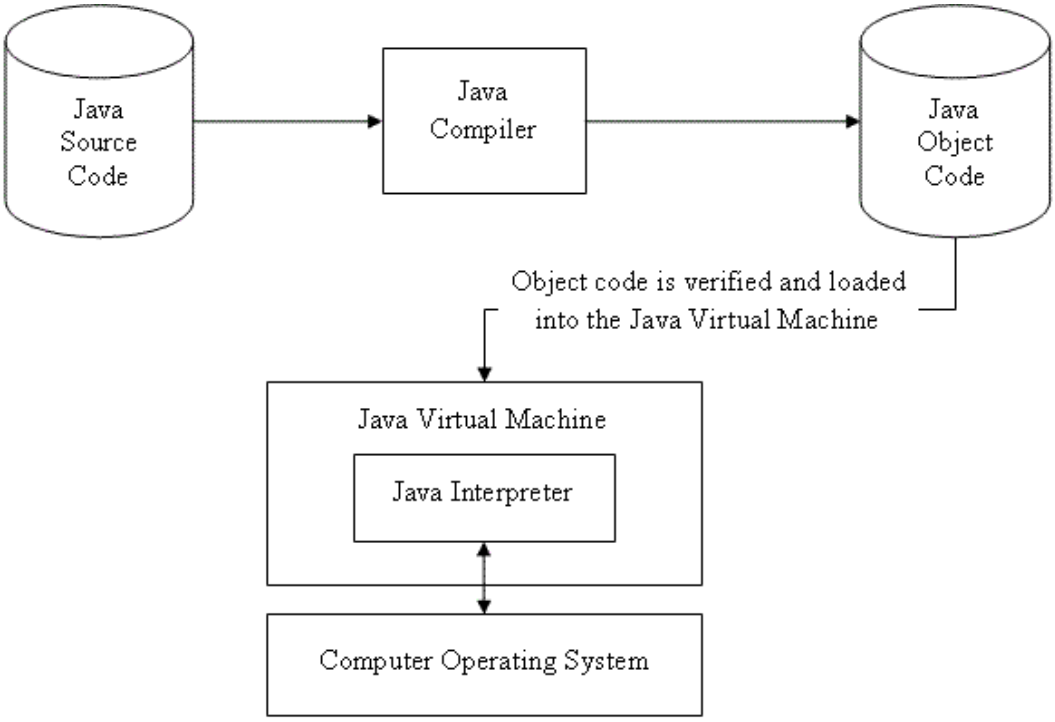

Figura 2. Abstracción de la máquina virtual de Java

Capítulo 2

Java Bytecode

2.1 Introducción

Como ya hemos visto en el capítulo anterior, un compilador realiza la función de traducir un lenguaje fuente en un lenguaje objeto, frente a los intérpretes, que ejecutan el código sin traducirlo a otro lenguaje intermedio.

Él código objeto del lenguaje de programación de la plataforma Java se denomina código de byte, o bytecode. El código recibe este nombre debido a que el cardinal de su conjunto de instrucciones es el valor de un byte, es decir, 256 instrucciones. Ésto quiere decir que la longitud del conjunto de instrucciones del bytecode es de un byte, pero la longitud del código de cada instrucción en particular es variable, como ya veremos más adelante.

La idea inicial de la especificación de este código sólo contemplaba que fuera a ser utilizada por la propia plataforma de Java. Sin embargo, debido a sus múltiples ventajas y a su gran compactibilidad, otros lenguajes de programación han sido diseñados para que tras ser compilados, produzcan bytecode que luego sea ejecutado por la máquina virtual de Java. Ejemplos de estos lenguajes son Perl, Phyton o PHP.

La funcionalidad que proporciona el juego de instrucciones del bytecode es extensa para ser un lenguaje objeto, ya que está orientado a la posibilidad de ser modificado “manualmente” por el programador, con lo cual el lenguaje alcanza un menor nivel de abstracción y es mucho más compacto. La máquina virtual de Java proporciona instrucciones cuyo objetivo es realizar las siguientes tareas:

- Aritméticas
- Carga y almacenamiento

- Creación y manipulación de objetos
- Transferencia de control
- Invocación y retorno a métodos
- Lanzamiento de excepciones
- Conversión de tipos
- Gestión de pilas

Ahora veremos más concretamente cada tipo de instrucción

2.2 Sintaxis

Para la especificación de la sintaxis del bytecode que vamos a describir a continuación, hemos tomado la versión 1.0.2 del Java Development Kit.

El formato de las instrucciones del bytecode está constituido de la siguiente manera:

<índice> <código de instrucción> [<operando1>[<operando2>….]] [<comentarios>]

De aquí podemos resaltar los siguientes campos:

1. <***índice***> es el índice de la operación, es decir, la posición que ocupa esa instrucción en el array de código que luego ejecutará la máquina virtual. Como ya hemos dicho antes, no todas las instrucciones tienen la misma longitud de código, cada instrucción tiene su propia longitud; es por esto que podemos encontrar instrucciones seguidas con un incremento de varias unidades en su índice y son generalmente las instrucciones de más complejidad: saltos, comparaciones en punto flotante….

   El campo índice también puede entenderse como el objetivo de las instrucciones de transferencia de control, realizando así la función de direcciones de memoria o etiquetas en otros lenguajes de bajo nivel; esto es, como el operando que indica el destino de un salto.

2. <***código de instrucción***> indica el código de operación, o lo que es lo mismo, el nombre de la instrucción correspondiente. Como ya hemos dicho, existen 256 códigos de operaciones distintos en la especificación completa del lenguaje.

3. Los comentarios vienen precedidos de dos barras “//”, al igual que, por ejemplo, en la sintaxis del lenguaje C/C++.

Para hacer sus cálculos correspondientes, el bytecode incorpora en su entorno de ejecución una pila de operandos y un conjunto de variables locales. Al ser el bytecode un lenguaje orientado a 3 direcciones de memoria, (Three-Address-Memory), la pila solo podrá operar, como mucho, con 2 operandos, y apilar el resultado, en contraposición con otros lenguajes imperativos, que pueden hacer cálculos apilado en su pila interna el números de operandos que sea, hasta que haya un desbordamiento de pila. Como veremos más adelante, esto

será una ventaja que nos servirá de gran ayuda para desarrollar el compilador, ya que en las reglas de la representación intermedia, los argumentos indican las variables de entrada, las de salida, las de pila y las de excepciones, por lo que los operandos de pila vendrán ya especificados y no tendremos que preocuparnos por asuntos como la asociatividad.

2.3 Tipos de operandos

Los tipos que puede manejar el bytecode son, básicamente, los tipos primitivos que manejan la mayoría de los lenguajes imperativos, tales como tipos numéricos o caracteres; y referencias a objetos. Más concretamente, los tipos son:

ENTEROS

byte – Se representan con 1 byte – Números de 0 a 255
short – Se representan con 2 bytes – Números de 0 a 65535
int – Se representan con 4 bytes – Números de 0 a 4 GB
long – Se representan con 8 bytes – Números de 0 a 16 TB

REALES

float – Se representan con 4 bytes usando coma flotante normalizada
double – Se representan con 8 bytes usando coma flotante normalizada

CARACTERES

char – Se representan con 1 byte – Los caracteres se codifican como números, usando el sistema de codificación Unicode.

REFERENCIAS A OBJETOS

2.4 Tipos de instrucciones

Como ya hemos dicho antes, el bytecode contempla 256 instrucciones distintas clasificadas en varios tipos. A continuación vamos a ver todas y cada una de ellas y su sintaxis. El repertorio de instrucciones está diseñado en su mayor parte para operaciones que manejan datos de tipo entero, dada su mayor frecuencia de aparición en programas típicos.

2.4.1 ARITMÉTICAS

Operan con los tipos primitivos numéricos:

Suma

iadd – Suma 2 enteros de la cima de la pila y apila el resultado
ladd – Suma 2 long de la cima de la pila y apila el resultado
fadd – Suma 2 float de la cima de la pila y apila el resultado
dadd – Suma 2 double de la cima de la pila y apila el resultado

Resta

isub – Resta 2 enteros de la cima de la pila y apila el resultado
lsub – Resta 2 long de la cima de la pila y apila el resultado
fsub – Resta 2 float de la cima de la pila y apila el resultado
dsub – Resta 2 double de la cima de la pila y apila el resultado

Multiplicación

imul – Multiplica 2 enteros de la cima de la pila y apila el resultado
lmul – Multiplica 2 long de la cima de la pila y apila el resultado
fmul – Multiplica 2 float de la cima de la pila y apila el resultado
dmul – Multiplica 2 double de la cima de la pila y apila el resultado

División

idiv – Divide 2 enteros de la cima de la pila y apila el resultado
ldiv – Divide 2 long de la cima de la pila y apila el resultado
fdiv – Suma 2 float de la cima de la pila y apila el resultado
ddiv – Suma 2 double de la cima de la pila y apila el resultado

Módulo

irem – Hace el módulo de 2 enteros de la cima de la pila y apila el resultado
lrem – Hace el módulo de 2 long de la cima de la pila y apila el resultado
frem – Hace el módulo de 2 float de la cima de la pila y apila el resultado
drem – Hace el módulo de 2 double de la cima de la pila y apila el resultado

AND Lógico

iand – Hace el AND lógico de 2 enteros de la cima de la pila y apila el resultado
land – Hace en AND lógico de 2 long de la cima de la pila y apila el resultado

OR Lógico

ior – Hace el OR lógico de 2 enteros de la cima de la pila y apila el resultado
lor – Hace en OR lógico de 2 long de la cima de la pila y apila el resultado

XOR Lógico

ixor – Hace el XOR lógico de 2 enteros de la cima de la pila y apila el resultado
lxor – Hace en XOR lógico de 2 long de la cima de la pila y apila el resultado

Operaciones aritméticas unarias

ineg – Hace la negación de un entero en la cima de la pila y apila el resultado
lneg – Hace la negación de un entero en la cima de la pila y apila el resultado
fneg – Hace la negación de un entero en la cima de la pila y apila el resultado
dneg – Hace la negación de un entero en la cima de la pila y apila el resultado

Operaciones de desplazamiento

ishl – Desplazamiento a la izquierda de un entero
ishr – Desplazamiento aritmético a la derecha de un entero
iushr – Desplazamiento lógico a la derecha de un entero
lshl – Desplazamiento a la izquierda de un long
lshr – Desplazamiento aritmético a la derecha de un long
lushr – Desplazamiento lógico a la derecha de un long

Operación de incremento

iinc <***variable***> <***valor***> - Incrementa la variable especificada el valor especificado sin tener que apilarla. Se atiende de manera especial por la máquina virtual de Java, y es muy útil en construcciones iterativas.

2.4.2 CARGA Y ALMACENAMIENTO

Las operaciones de carga leen el contenido de una variable local especificada en la propia instrucción y lo apilan en la cima. También es posible apilar una serie de valores directamente, sin necesidad de que hayan sido previamente almacenados en una variable local. Son los valores -1, 0, 1, 2 , 3 ,4 , y 5. Tiene lugar este uso de determinados operandos implícitos por cuestión de alta frecuencia de aparición de estos valores.

Las operaciones de almacenamiento escriben el valor ubicado en la cima de la pila en una variable local que nosotros le indiquemos.

Es necesario recordar que los valores de tipo long y de tipo double, que son los que se implementan con una longitud de 8 bytes, ocupan dos variables locales, aunque sólo son accesibles utilizando el índice más pequeño.

iload_0 – Carga en la pila el valor entero de la variable local 0
iload_1 – Carga en la pila el valor entero de la variable local 1
iload_2 – Carga en la pila el valor entero de la variable local 2
iload_3 – Carga en la pila el valor entero de la variable local 3
iload n – Carga en la pila el valor entero de la variable local n

lload_0 – Carga en la pila el valor long de la variable local 0
lload_1 – Carga en la pila el valor long de la variable local 1
lload_2 – Carga en la pila el valor long de la variable local 2
lload_3 – Carga en la pila el valor long de la variable local 3
lload n – Carga en la pila el valor long de la variable local n

fload_0 – Carga en la pila el valor float de la variable local 0
fload_1 – Carga en la pila el valor float de la variable local 1
fload_2 – Carga en la pila el valor float de la variable local 2
fload_3 – Carga en la pila el valor float de la variable local 3
fload n – Carga en la pila el valor float de la variable local n

dload_0 – Carga en la pila el valor double de la variable local 0
dload_1 – Carga en la pila el valor double de la variable local 1
dload_2 – Carga en la pila el valor double de la variable local 2
dload_3 – Carga en la pila el valor double de la variable local 3
dload n – Carga en la pila el valor double de la variable local n

aload_0 – Carga en la pila la referencia al objeto de la variable local 0
aload_1 – Carga en la pila la referencia al objeto de la variable local 1
aload_2 – Carga en la pila la referencia al objeto de la variable local 2
aload_3 – Carga en la pila la referencia al objeto de la variable local 3
aload n – Carga en la pila la referencia al objeto de la variable local n

istore_0 – Almacena en la variable local 0 el entero de la cima de la pila
istore_1 – Almacena en la variable local 1 el entero de la cima de la pila
istore_2 – Almacena en la variable local 2 el entero de la cima de la pila
istore_3 – Almacena en la variable local 3 el entero de la cima de la pila
istore n – Almacena en la variable local n el entero de la cima de la pila

lstore_0 – Almacena en la variable local 0 el long de la cima de la pila
lstore_1 – Almacena en la variable local 1 el long de la cima de la pila
lstore_2 – Almacena en la variable local 2 el long de la cima de la pila
lstore_3 – Almacena en la variable local 3 el long de la cima de la pila
lstore n – Almacena en la variable local n el long de la cima de la pila

fstore_0 – Almacena en la variable local 0 el float de la cima de la pila
fstore_1 – Almacena en la variable local 1 el float de la cima de la pila
fstore_2 – Almacena en la variable local 2 el float de la cima de la pila
fstore_3 – Almacena en la variable local 3 el float de la cima de la pila
fstore n – Almacena en la variable local n el float de la cima de la pila

dstore_0 – Almacena en la variable local 0 el double de la cima de la pila
dstore_1 – Almacena en la variable local 1 el double de la cima de la pila
dstore_2 – Almacena en la variable local 2 el double de la cima de la pila
dstore_3 – Almacena en la variable local 3 el double de la cima de la pila
dstore n – Almacena en la variable local n el double de la cima de la pila

astore_0 – Almacena en la variable local 0 la referencia al objeto de la cima de la pila

astore_1 – Almacena en la variable local 1 la referencia al objeto de la cima de la pila
astore_2 – Almacena en la variable local 2 la referencia al objeto de la cima de la pila
astore_3 – Almacena en la variable local 3 la referencia al objeto de la cima de la pila
astore n – Almacena en la variable local n la referencia al objeto de la cima de la pila

iconst_0 – Apila el valor entero constante 0
iconst_1 – Apila el valor entero constante 1
iconst_2 – Apila el valor entero constante 2
iconst_3 – Apila el valor entero constante 3
iconst_4 – Apila el valor entero constante 4
iconst_5 – Apila el valor entero constante 5
iconst_m1 – Apila el valor entero constante -1

lconst_0 – Apila el valor long constante 0
lconst_1 – Apila el valor long constante 1

fconst_0 – Apila el valor float constante 0.0
fconst_1 – Apila el valor float constante 1.0
fconst_2 – Apila el valor float constante 2.0

dconst_0 – Apila el valor double constante 0.0
dconst_1 – Apila el valor double constante 1.0

bipush n – Apila el valor entero n, 0 <= n <=255
sipush n – Apila el valor entero n, 0 <= n <= 65535

ldc <***valor***> - Apila el valor entero largo o referencia a String
ldc_w <***valor***> - Apila el valor entero largo o referencia a String, cuando se necesita un índice largo
ldc2_w <***valor***> - Apila el valor float o double

aconst_null – Apila la referencia al objeto nulo (***null***)

baload – Apila un valor de tipo byte contenido en un array
caload – Apila un valor de tipo char contenido en un array
saload – Apila un valor de tipo short contenido en un array
iaload – Apila un valor de tipo entero contenido en un array
laload – Apila un valor de tipo long contenido en un array
faload – Apila un valor de tipo float contenido en un array
daload – Apila un valor de tipo double contenido en un array
aaload – Apila una referencia a un objeto contenida en un array

bastore – Almacena la cima de la pila en un array de tipo byte
castore – Almacena la cima de la pila en un array de tipo char
sastore – Almacena la cima de la pila en un array de tipo short
iastore – Almacena la cima de la pila en un array de tipo entero

lastore – Almacena la cima de la pila en un array de tipo long
fastore – Almacena la cima de la pila en un array de tipo float
dastore – Almacena la cima de la pila en un array de tipo double
aastore – Almacena la cima de la pila en un array de referencias a objetos

2.4.3 CREACIÓN Y MANIPULACIÓN DE OBJETOS

Manipulación de objetos

new <***nombre de la clase***> - Crea un objeto de una clase
instanceof <***nombre de la clase***> - Determina si el objeto es una instancia de la clase
checkcast <***tipo***> - Comprueba si el objeto es del tipo especificado

Manipulación de atributos

getfield <***nombre del atributo***> - Apila el valor del atributo especificado
putfield <***nombre del atributo***> - Almacena el valor de la pila en el atributo
getstatic <***nombre del atributo***> - Apila el valor del atributo estático especificado
putstatic <***nombre del atributo***> – Almacena el valor de la pila en el atributo estático

Creación y manipulación de arrays

Los arrays también son tratados como referencias a objetos

newarray <***tipo***> - Crea un array del tipo especificado
anewarray <***nombre de la clase***> - Crea un array de referencias a objetos de la clase especificada
multianewarray <***tipo***> <***dimensión***> - Crea un array multidimensional del tipo especificado con las dimensiones especificadas
arraylength – Devuelve la longitud de un array

2.4.4 TRANSFERENCIA DE CONTROL

icmpeq <***dir***> - Salta a la dirección especificada si los dos enteros de la cima de la pila son iguales
icmpne <***dir***> - Salta a la dirección especificada si los dos enteros de la cima de la pila no son iguales
icmpge <***dir***> - Salta a la dirección especificada si el entero de la cima de la pila es mayor o igual que el que está por debajo de la cima
icmpgt <***dir***> - Salta a la dirección especificada si el entero de la cima de la pila es mayor que el que está por debajo de la cima
icmple <***dir***> - Salta a la dirección especificada si el entero de la cima de la pila es menor o igual que el que está por debajo de la cima
icmplt <***dir***> - Salta a la dirección especificada si el entero de la cima de la pila es menor que el que está por debajo de la cima

ifeq <***dir***> - Salta a la dirección especificada si el valor entero del operando de la cima de la pila es igual a cero
ifne <***dir***> - Salta a la dirección especificada si el valor entero del operando de la cima de la pila no es igual a cero
ifge <***dir***> - Salta a la dirección especificada si el valor entero del operando de la cima de la pila es mayor o igual que cero.
ifgt <***dir***> - Salta a la dirección especificada si el valor entero del operando de la cima de la pila es mayor que cero.
ifle <***dir***> - Salta a la dirección especificada si el valor entero del operando de la cima de la pila es menor o igual que cero.
iflt <***dir***> - Salta a la dirección especificada si el valor entero del operando de la cima de la pila es menor que cero.
ifnull <***dir***> - Salta a la dirección especificada si la referencia de la cima de la pila es el objeto nulo
ifnonnull <***dir***> - Salta a la dirección especificada si la referencia de la cima de la pila no es el objeto nulo

Las siguientes instrucciones lo que hacen es comparar que el valor apilados sea de un determinado tipo, ya que en el caso de long, float y double no existen instrucciones directas de comparación, como es el caso de los enteros. Para comparar estos valores, se utilizara una instrucción de comparación a cero de las especificadas anteriormente y se hará un chequeo del tipo. Tanto las instrucciones que chequean el tipo float como las que chequean el tipo double tienen dos posibles instrucciones para hacer esta comprobación; sin embargo, en lo único en lo que difieren es en el uso que se les da a los valores no normalizados (***NaN***).

fcmpl – Compara que el valor sea de tipo float
fcmpg – Compara que el valor sea de tipo float
dcmpl – Compara que el valor sea de tipo double
dcmpg – Compara que el valor sea de tipo double
lcmp – Compara que el valor sea de tipo long

goto <***dir***> – Salta a la dirección especificada
goto_w <***dir***> – Salta a la dirección especificada (con el índice más ancho posible, la resolución del computador)

Un caso especial son los switches, en los que es necesario construir una tabla de saltos. La instrucción ***tableswitch*** se usa cuando los índices se pueden considerar como un rango ordenado, mientras que la instrucción ***lookupswitch*** acepta cualquier tipo de rango.

tableswitch <***índice_inicial***> ***to*** <***índice_final***>
<***índice1***>: <***dir1***>
<***índice2***>: <***dir2***>

<***índicenN***>: <***dirN***>
Default: <***dirAlternativa***>

Cuando los casos del switch son escasos, la tabla de saltos de la instrucción ***tableswitch*** se vuelve ineficiente en términos de espacio. La instrucción

lookupswitch empareja elementos de la forma <clave, desplazamiento>, donde la clave será el índice del case, y el desplazamiento la instrucción destino del salto.

lookupswitch *<número de índices>*
<índice1>: <desplazamiento1>
<índice2>: <desplazamiento2>

<índiceN>: <desplazamientoN>
Default: <deplazamientoAlternativo>

2.4.5 INVOCACIÓN Y RETORNO A MÉTODOS

En el bytecode, y en otros muchos lenguajes de programación, cada vez que invocamos a un método creamos un nuevo marco de activación y cambiamos el contexto a ese marco de activación; es decir, creamos una nueva pila de operandos y un nuevo conjunto de variables locales para ese método en particular, y ambos elementos serán destruidos una vez haya acabado la ejecución de éste.

De esta manera, cuando existan varios métodos anidados, coexistirán varios marcos de activación simultáneamente, uno para cada uno. Sin embargo, sólo estará activa la pila de operandos del marco actual.

A un nivel más interno, los parámetros de las invocaciones a métodos son pasados como variables locales desde 1 hasta n, siendo n el número de parámetros, ya que la variable local 0 está reservada para la referencia ***"this"*** de la invocación al método. En el caso de métodos estáticos, como no es posible su instanciación, se empezaran a designar sus parámetros desde la variable local 0.

invokevirtual *<nombre del método>* - Invoca el método especificado
invokespecial *<nombre del método>* - Invoca el método especificado, siempre que éste sea el constructor de una clase, un método privado o el método que invoca a un método de la superclase
invokestatic *<nombre del método>* - Invoca el método estático especificado
invokeinterface *<nombre de la interfaz>* - Invoca un método de la interfaz especificada.

return – No retorna ningún valor
ireturn – Retorna un valor de tipo entero de la cima de la pila del método
lreturn – Retorna un valor de tipo long de la cima de la pila del método
freturn – Retorna un valor de tipo float de la cima de la pila del método
dreturn – Retorna un valor de tipo double de la cima de la pila del método
areturn – Retorna una referencia a un objeto de la cima de la pila del método

2.4.6 Conversión de tipos

i2s – Hace casting de entero a short
i2b – Hace casting de entero a byte
i2c – Hace casting de entero a char
i2l – Hace casting de entero a long
i2f – Hace casting de entero a float
i2d – Hace casting de entero a double
l2i – Hace casting de long a entero
l2f – Hace casting de long a float
l2d – Hace casting de long a double
f2i – Hace casting de float a entero
f2l – Hace casting de float a long
f2d – Hace casting de float a double
d2i – Hace casting de double a entero
d2l – Hace casting de double a long
d2f – Hace casting de double a float

2.4.7 Gestión de pilas

Otra funcionalidad que nos proporciona el bytecode es manipular los contenidos de la pila de operandos como valores no tipados. Sin embargo, lo que no permite es la modificación o eliminación de valores individuales.

dup – Duplica el valor de la cima de la pila
dup_x1 – Duplica el valor de la cima de la pila insertando dos valores debajo de él
dup_x2 – Duplica el valor de la cima de la pila insertando dos o tres valores debajo de él
dup2 – Duplica uno o dos valores de la cima de la pila
dup2_x1 – Duplica uno o dos valores de la cima de la pila insertando dos o tres valores debajo de ellos
dup2_x2 – Duplica uno o dos valores de la cima de la pila insertando dos, tres o cuatro valores debajo de ellos
pop – Elimina el valor de la cima de la pila
pop2 – Elimina uno o dos valores de la cima de la pila
swap – Intercambia los dos valores de la cima de la pila
wide – Extiende el operando al máximo ancho de posible
nop – No hace nada

2.4.8 Sincronización

Otra de las múltiples funcionalidades que permite el bytecode, es soporte específico para la sincronización de programas.

monitorenter – A partir de esta instrucción el thread en uso adquiere el monitor

monitorexit – A partir de esta instrucción el thread en uso libera el monitor

A pesar de proporcionar éstas dos instrucciones, su uso no es la manera habitual de sincronizar threads en bytecode, ya que cuenta con un flag especial, llamado ***ACC_SYNCHRONIZED***, que se comprueba en cada invocación a método, de forma que estando activado, ese thread adquirirá el monitor y posteriormente lo liberará por si mismo.

2.5. Excepciones

Una característica especial que permite el bytecode es el manejo de excepciones. Para ello, mantiene una tabla de excepciones, en la que queda constancia del tipo de excepción lanzada (o bien, si es la clase genérica ***Exception*** almacenaremos el nombre de esa clase); del rango de índices de instrucciones del bytecode en el que puede producirse la excepción (en código fuente Java sería equivalente a la porción de código comprendido entre la cláusula ***try-catch***), guardando tanto la dirección inicial del rango como la final; y por último también guardamos el índice al que debemos saltar si durante la ejecución del programa la excepción realmente se produce. Esta última dirección se corresponde con el manejador de la excepción, es decir, el código de la cláusula ***catch***.

Durante la ejecución del bytecode si se produce una excepción se invocará al constructor de la clase que implemente el tipo de la excepción que haya tenido lugar, y se invoca como si fuera cualquier otro método ordinario; en este caso, como es un constructor, con la instrucción ***invokespecial***.
De esta manera, el compilador de Java introduce una nueva entrada en la tabla de excepciones por cada manejador implementado en el código fuente; si éstos no existieran, la tabla no sería construida a pesar de todas las posibles excepciones que puedan tener lugar.

La tabla de excepciones tiene la forma

<***nombre del método***>
Indice inicial ***Índice final*** ***Destino*** ***Tipo de excepción***

Las instrucciones del código de byte relacionadas con el manejo de excepciones son las siguientes.

athrow – Lanza una excepción
jsr <***dir***> – Salta a la subrutina que se encuentra en la dirección especificada
jsr_w <***dir***> – Salta a la subrutina que se encuentra en la dirección especificada (índice ancho)
ret <***dir***> - Retorna desde la subrutina a la dirección especificada para seguir el flujo de control principal

2.6 Constant pool

Aparte del conjunto de instrucciones y de la tabla de excepciones, un fichero ***.class*** proporciona otros datos. Todas las instrucciones que se han explicado anteriormente, no son como realmente las produce el compilador de Java. En todas aquellas que contengan algún argumento no implícito, éste será sustituido por una referencia simbólica generada en tiempo de ejecución. El conjunto de todas estas referencias es lo que se denomina el constant pool. Los literales que almacena pueden ser de varios tipos, como Strings, identificadores, referencias y clases y métodos, y descriptores de tipos. Todos estos tipos quedan referenciados mediante una constante específica, y son codificados en el constant pool como números de 16 bits. Los tipos long y double ocupan dos entradas en el constant pool, aunque el segundo índice nunca será diseccionado. Las entradas de este conjunto empiezan a numerarse desde el valor constante 1, ya que el 0 es inválido, está reservado para la propia clase ***Object***.

2.7 Otros metadatos

Un archivo ***.class***, además de contener un programa en bytecode y el conjunto de referencias simbólicas, presenta otros metadatos que necesitaremos conocer para finalizar la traducción.

Examinando la estructura interna de un archivo ***.class***, se encuentran otras secciones básicas aparte del constant pool.

- El número mágico, que es el ya conocido ***0xCAFEBABE***
- La versión del archivo .***class***, que consiste en las versiones mayor y menor del fichero
- El nombre de la propia clase, la superclase, las interfaces que implementa, los atributos, los métodos, los modificadores de acceso, y otros atributos, como por ejemplo el nombre del fichero fuente.

Además, al desensamblar un fichero ***.class*** con la utilidad ***javap***, se pueden observar más metadatos internos a cada método:

- Tamaño de pila: Si el tamaño de pila es menos del que necesitaremos durante la ejecución del programa, el verificador de la máquina virtual de Java lanza un error en tiempo de ejecución informándonos de que el tamaño de pila es demasiado pequeño.
- Tamaño del conjunto de variables locales: El número de variables locales que se utilizarán en cada método.
- Número de argumentos del método

2.8 Ejemplo

A continuación se va a mostrar un ejemplo de un programa real en bytecode, mostrando primero el código fuente original en Java, y posteriormente toda la información del fichero ***.class*** que resultaría de su compilación, tanto el código de byte, como el constant pool y el resto de metadatos necesarios para su ejecución en la máquina virtual de Java.

El programa está diseñado con un código muy básico. Las clases, atributos y métodos no están dotados de directivas de control de acceso, haciéndose de esta manera visibles en todo el paquete. La clase ***test*** tiene un constructor en el que se llama recursivamente a la clase ***Tree***, definida posteriormente. En el constructor también se llaman a los métodos estáticos f y g. En este ejemplo lo que mejor se va a poder identificar son las llamadas a métodos y el manejo de clases y atributos, ya que al ser un ejemplo sencillo, no se van a generar gran parte de los códigos de operaciones del bytecode que han sido mencionados anteriormente, como de gestión de pila, de conversión de tipos o de sincronización.

```
package misc.students;

	class test {
		Tree m(int n) {
			if ( n>0 ) return new Tree(m(n-1),m(n-1),f(n));
			else return null;
		}

		static int f(int n) {
			int a=0;
			int i=n;

			while ( n>1 ) {
				a += g(new Integer(n));
				n=n/2;
			}

			for(; i>0; i--)
				a *=  g(new Integer(i));

			return a;
		}

		static int g(Integer n) {
			return n.intValue()+1;
		}

	} // end of class test

	class Tree {
		Tree l;
		Tree r;
		int  d;

		Tree(Tree l,Tree r,int d) {
			this.l = l;
			this.r = r;
			this.d = d;
		}
	}
```

Figura 3. Código fuente de las clases **misc/student/test** y **misc/students/Tree**

Una vez hemos compilado el ejemplo obtenemos un fichero ***.class***. Tras vislumbrar su contenido desensamblándolo con la utilidad ***javap,*** éste es el bytecode resultante.

Notar que al compilar un archivo ***.java***, en este caso, mediante línea de comandos

```
Compiled from "ex1.java"
class misc.students.test extends java.lang.Object
  SourceFile: "ex1.java"
  minor version: 0
  major version: 0
  Constant pool:
const #1 = Method       #11.#24;        //  java/lang/Object."<init>":()V
const #2 = class        #25;    //  Tree
const #3 = Method       #10.#26;        //  misc/students/test.m:(I)Lmisc/studen
ts/Tree;
const #4 = Method       #10.#27;        //  misc/students/test.f:(I)I
const #5 = Method       #2.#28; //  misc/students/Tree."<init>":(Lmisc/students/
Tree;Lmisc/students/Tree;I)V
const #6 = class        #29;    //  Integer
const #7 = Method       #6.#30; //  java/lang/Integer."<init>":(I)V
const #8 = Method       #10.#31;        //  misc/students/test.g:(Ljava/lang/Int
eger;)I
const #9 = Method       #6.#32; //  java/lang/Integer.intValue:()I
const #10 = class       #33;    //  test
const #11 = class       #34;    //  Object
const #12 = Asciz       <init>;
const #13 = Asciz       ()V;
const #14 = Asciz       Code;
const #15 = Asciz       LineNumberTable;
const #16 = Asciz       m;
const #17 = Asciz       (I)Lmisc/students/Tree;;
const #18 = Asciz       f;
const #19 = Asciz       (I)I;
const #20 = Asciz       g;
const #21 = Asciz       (Ljava/lang/Integer;)I;
const #22 = Asciz       SourceFile;
const #23 = Asciz       ex1.java;
const #24 = NameAndType #12:#13;//  "<init>":()V
const #25 = Asciz       misc/students/Tree;
const #26 = NameAndType #16:#17;//  m:(I)Lmisc/students/Tree;
const #27 = NameAndType #18:#19;//  f:(I)I
const #28 = NameAndType #12:#35;//  "<init>":(Lmisc/students/Tree;Lmisc/students
/Tree;I)V
const #29 = Asciz       java/lang/Integer;
const #30 = NameAndType #12:#36;//  "<init>":(I)V
const #31 = NameAndType #20:#21;//  g:(Ljava/lang/Integer;)I
const #32 = NameAndType #37:#38;//  intValue:()I
const #33 = Asciz       misc/students/test;
const #34 = Asciz       java/lang/Object;
const #35 = Asciz       (Lmisc/students/Tree;Lmisc/students/Tree;I)V;
const #36 = Asciz       (I)V;
const #37 = Asciz       intValue;
const #38 = Asciz       ()I;

{
misc.students.test();
```

```
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return
  LineNumberTable:
   line 2: 0

misc.students.Tree m(int);
  Code:
   Stack=6, Locals=2, Args_size=2
   0:   iload_1
   1:   ifle    30
   4:   new     #2; //class Tree
   7:   dup
   8:   aload_0
   9:   iload_1
   10:  iconst_1
   11:  isub
   12:  invokevirtual   #3; //Method m:(I)Lmisc/students/Tree;
   15:  aload_0
   16:  iload_1
   17:  iconst_1
   18:  isub
   19:  invokevirtual   #3; //Method m:(I)Lmisc/students/Tree;
   22:  iload_1
   23:  invokestatic    #4; //Method f:(I)I
   26:  invokespecial   #5; //Method misc/students/Tree."<init>":(Lmisc/students
/Tree;Lmisc/students/Tree;I)V
   29:  areturn
   30:  aconst_null
   31:  areturn
  LineNumberTable:
   line 4: 0
   line 6: 30

static int f(int);
  Code:
   Stack=4, Locals=3, Args_size=1
   0:   iconst_0
   1:   istore_1
   2:   iload_0
   3:   istore_2
   4:   iload_0
   5:   iconst_1
   6:   if_icmple       30
   9:   iload_1
   10:  new     #6; //class Integer
   13:  dup
   14:  iload_0
   15:  invokespecial   #7; //Method java/lang/Integer."<init>":(I)V
   18:  invokestatic    #8; //Method g:(Ljava/lang/Integer;)I
   21:  iadd
   22:  istore_1
   23:  iload_0
   24:  iconst_2
   25:  idiv
   26:  istore_0
   27:  goto    4
   30:  iload_2
   31:  ifle    54
   34:  iload_1
   35:  new     #6; //class Integer
   38:  dup
   39:  iload_2
   40:  invokespecial   #7; //Method java/lang/Integer."<init>":(I)V
   43:  invokestatic    #8; //Method g:(Ljava/lang/Integer;)I
```

```
   46:  imul
   47:  istore_1
   48:  iinc    2, -1
   51:  goto    30
   54:  iload_1
   55:  ireturn
  LineNumberTable:
   line 10: 0
   line 11: 2
   line 13: 4
   line 14: 9
   line 15: 23
   line 18: 30
   line 19: 34
   line 18: 48
   line 21: 54

static int g(java.lang.Integer);
  Code:
   Stack=2, Locals=1, Args_size=1
   0:   aload_0
   1:   invokevirtual   #9; //Method java/lang/Integer.intValue:()I
   4:   iconst_1
   5:   iadd
   6:   ireturn
  LineNumberTable:
   line 24: 0

}
```

Figura 4. Bytecode de la clase misc/students/test

Una vez obtenido el contenido de la compilación, vamos a resaltar varios puntos.

Al principio de cada fichero, siempre obtenemos información del fichero fuente, y las versiones mayor y menor del fichero.

Las signaturas de clases, métodos y atributos se especifican en el bytecode con el nombre del paquete seguido del nombre de la clase, y seguido del nombre del atributo o método en cuestión en el caso de ser manejado uno de éstos.

Cada clase tiene su propia información acerca de las clases que extiende y de las interfaces que implementa; en este caso, como no hereda de ninguna clase ni implementa ninguna interfaz, sólo se indica que extiende la clase ***Object***.

A continuación se muestra el constant pool, en el que se puede observar cómo el compilador de Java ha establecido todas las referencias simbólicas (al igual que, como veremos más adelante realiza la aplicación ***Jasmin***, en el objetivo principal de este proyecto). Esas referencias pueden apuntar a múltiples tipos de objetos, y también, con mucha frecuencia, pueden apuntar sucesivamente a otras referencias, hasta hacer visible su contenido final. Siguiendo este razonamiento, se puede ver que la referencia simbólica ***const #2*** apunta a otra referencia, ***const #25***, y ésta apunta a la clase ***misc/students/Tree***. Toda esta heterogeneidad de posibles elementos referenciados se indica en primer

parámetro de cada entrada del constant pool, en este ejemplo toma valores como ***Class***, ***Method***, ***Asciz***….

Al principio de cada método figuran los metadatos locales, como ya hemos visto, el tamaño de la pila, el número de variables locales y el tamaño del array de argumentos (toda esta información se guardará al hacer la decompilación al formato de reglas figurando en la cabecera de cada una de ellas).

Para facilitar la comprensión del código, se especifica al lado de cada referencia simbólica el elemento apuntado.

Obviamos las directivas ***line*** y ***LineNumberTable*** ya que únicamente se usa con perspectivas a la depuración de código.

Capítulo 3

Especificación de la representación intermedia basada en reglas

A partir del bytecode, mediante el uso de técnicas básicas usadas en el ámbito de compilación [*Aho et al. 1974; Aho et al. 1986*], ***COSTA*** genera un grafo de control de flujo de ejecución para cada uno de los métodos de la clase que tiene como entrada.

Cada uno de estos grafos de control se representa mediante un conjunto de procedimientos, mediante una representación basada en reglas recursivas (***rbr-rules***).

El proyecto trabaja sobre esta representación intermedia, que a continuación es especificada como gramática de manera simplificada, pues hay muchos campos en las reglas, que no son utilizados a la hora de realizar el proceso de traducción a bytecode.

Regla → ***Id := [Guarda][Bytecode] * [CallMethod][CallBlock]***

Id → ***Signatura de Método***
Id → ***Numero Natural***

La primera de las producciones es la referente al identificador de la primera regla, que corresponde a la signatura del método, con información del paquete, clase y nombre. La segunda es un identificador que tiene la finalidad de identificar bloques y dirigir el flujo hacia uno u otro bloque, con la finalidad de representar bloques intermedios. Más adelante en un ejemplo se especifica el funcionamiento mediante un gráfico.

Guarda → ***assr (condición)***

Para los conjuntos de reglas que comparten identificador, para seleccionar una u otra a la hora de transitar en el grafo, se usa una guarda en la que se comprueba una condición. Sólo una de las guardas es cierta, para garantizar un flujo correcto. Existen condiciones de varios tipos:

- Comprobación relativa a 0 del valor de la pila de operandos, para instrucciones de salto.
- Comprobación de si la cima de la pila es un valor nulo.
- Comprobación de caso en un ***switch***.
- Comprobaciones de clase mediante ***instanceof***, usadas sobre todo para comprobar el tipo de excepciones.

Bytecode* → *bytecode* *(índice, instrucción)

Cero o más bytecodes que representan bloques de instrucciones en este lenguaje, donde ***índice*** es la posición del bytecode en el programa original, e ***instrucción*** corresponde a la denominación del bytecode. Existe una denominación para cada una de las 256 instrucciones que componen el bytecode, y están todas definidas en el capitulo anterior.

CallMethod* → *call* *(method (Tipo) , NombreMétodo)

En el conjunto de instrucciones del bloque de una regla, puede haber una llamada a método. ***Tipo*** indica si se refiere a un método de tipo virtual, especial o estático.

CallBlock* → *call* *(Número natural)

Llamada a la regla siguiente.

Continuando con el ejemplo visto en el apartado anterior, a continuación se muestran las reglas intermedias generadas por el decompilador para el método ***m*** de la clase ***misc/students/Tree***:

```
misc/students/test_m(I)Lmisc/students/Tree:=
bytecode(-1, init_vars)
call(block, 1)

1:=
bytecode(0, push(i, 1))
bytecode(1, store(i, 2))
bytecode(2, load(i, 2))
nop(bytecode(3, lookupswitch(64, [(1, 44),(3, 54),(4, 59),(123, 49)]))
call(block, 19)

19 :=
assr(switch(l_succ, 1))
call(block, 2)
19:=
assr(switch(l_succ, 3))
call(block, 3)
19:=
assr(switch(l_succ, 4))
call(block, 4)
19:=
```

```
assr(switch(l_succ, 123))
call(block, 5)


19:=
assr(switch(l_def, [123, 4, 3, 1]))
call(block, 6)

2:=
bytecode(44, push(i, 1))
bytecode(45, store(i, 2))
bytecode(46, goto(66))
call(block, 7)

3:=
bytecode(54, push(i, 3))
bytecode(55, store(i, 2))
bytecode(56, goto(66))
call(block, 7)

4:=
bytecode(59, push(i, 4))
bytecode(60, store(i, 2))
nop(bytecode(61, goto(66))
call(block, 7)

5:=
bytecode(49, push(i, 2))
bytecode(50, store(i, 2))
nop(bytecode(51, goto(66))
call(block, 7)

6:=
bytecode(64, push(i, 5))
bytecode(65, store(i, 2))
call(block, 7)

20:=
assr(cmpz(le), [s(1)], [])
call(block, 8)
20:=
assr(cmpz(gt), [s(1)], [])
call(block, 9)

7:=
bytecode(66, load(i, 1))
bytecode(67, cmpz(le, 96))
call(block, 20)

8:=
bytecode(96, aconst_null)
bytecode(97, return(a))
call(block, 10)

9:=
bytecode(70, new(misc/students/Tree))
bytecode(73, dup)
bytecode(74, load(a, 0))
bytecode(75, load(i, 1))
bytecode(76, push(i, 1))
bytecode(77, barithm(i, sub))
bytecode(78, invoke(virtual, m(I)Lmisc/students/Tree))
call(block, 13)

13:=
bytecode(81, load(a, 0))
bytecode(82, load(i, 1))
```

```
bytecode(83, push(i, 1))
bytecode(84, barithm(i, sub))
bytecode(85, invoke(virtual,m(I)Lmisc/students/Tree))
call(block, 15)

15:=
bytecode(88, load(i, 1))
bytecode(89, invoke(static, misc/students/test_f(I)I))
call(method(static), misc/students/test_f(I)I)
call(block, 17)

17:=
bytecode(92, invoke(special, methodref(misc/students/Tree_<init>)))
call(method(special), misc/students/Tree_<init>)
call(block, 18)

18:=
bytecode(95, return(a))
call(block, 10)

10:=
```

Figura 5. Representación de reglas del método **m** de la clase **misc/students/Tree**

A continuación se presenta un diagrama que muestra el grafo de flujo de ejecución que representan las reglas, donde la etiqueta de cada rectángulo se refiere al identificador de la regla, y la etiqueta de cada rombo se refiere a las reglas múltiples, donde dependiendo de una condición se elije un camino u otro en el grafo.

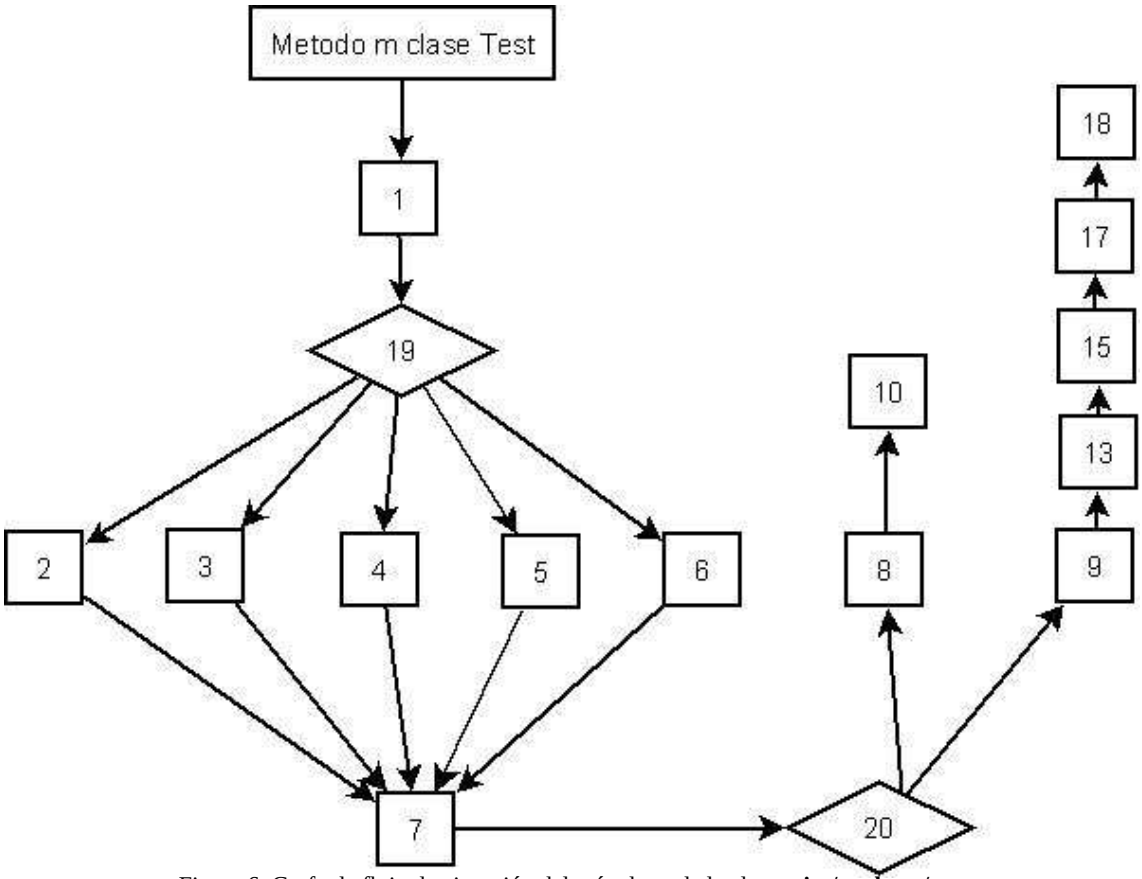

Figura 6. Grafo de flujo de ejecución del método m de la clase **misc/students/test**

Capítulo 4

Jasmin

4.1 Introducción

Jasmin es un ensamblador cuya función es crear archivos ***.class*** desde ficheros escritos en una notación propia, cercana al bytecode, pero sin referencias simbólicas y con un conjunto de directivas nativas de la propia aplicación. Esos nuevos ficheros .***class*** podrán ser ejecutados normalmente por la máquina virtual de Java.

La motivación de la creación de ***Jasmin*** es decrementar el nivel de dificultad para crear archivos ***.class***, haciendo más sencilla la exploración de las posibilidades que ofrece la máquina virtual de Java, y creando programas en un nuevo lenguaje que pueda etiquetar las direcciones de la máquina virtual, sin necesidad de tener conocimiento de las referencias del constant pool o de las tablas de atributos.

Es una aplicación de libre distribución, de código abierto, disponible bajo la licencia GNU/GPL, y en ningún caso su notación podría considerarse un lenguaje de programación.

A pesar de que la sintaxis de prácticamente la totalidad de las instrucciones del bytecode es similar a la notación de ***Jasmin***, hay algunos cambios significativos que merece la pena mencionar.

Al estar ***Jasmin*** programado en Java, es una aplicación multiplataforma, si bien, el comando para ejecutarlo cambia de Windows a Linux

Para ejecutarlo no hay más que escribir en la línea de comandos

java –jar jasmin.jar ***<nombre del archivo .j>***

Una vez ejecutado ***Jasmin*** su ensamblador informará de todos los errores sintácticos, si es que éstos se han producido. En caso de no haber ninguno, obtendremos como salida un fichero ***.class***. (Lo que no quiere decir que no existan errores de ejecución, como veremos más adelante)

4.2 Sintaxis

La sintaxis de ***Jasmin*** está compuesta de directivas, instrucciones y etiquetas.

4.2.1 DIRECTIVAS

Las directivas son las encargadas de proporcionar los metadatos al futuro archivo ***.class***, ya que se encargan de informar de la existencia de clases, atributos, métodos, excepciones, tamaño de pila…

Todas las directivas empiezan con un punto, y son las siguientes:

.source <***nombre del archivo***> - Indica el archive fuente desde el que se va a generar el ***.class***, suele ser el propio archivo .j
.super <***nombre de la clase***> - Indica la clase de la que hereda, si no hereda de ninguna se indica la clase ***java/lang/Object***
.class <***modificadores de acceso***> <***nombre de la clase***> - Indica el nombre de la clase y el tipo de acceso que tiene
.interface <***modificadores de acceso***> <***nombre de la interfaz***> - Indica el nombre de la interfaz y el tipo de acceso que tiene
.implements <***nombre de la interfaz***> - Indica la interfaz que implementa la clase, en caso de implementar alguna
.field <***modificadores de acceso***> <***nombre del atributo***> <***tipo***> - indica el nombre del atributo, su tipo, y el tipo de acceso que tiene. En caso de que esté declarado como final, puede inicializarse con su valor correspondiente.
.method <***modifcadores de acceso***> <***nombre del metodo***> - Indica el nombre del método y el tipo de acceso que tiene
.limit <***elemento***> <***tamaño***> - Hay dos posibles opciones de uso de la directiva ***.limit***; para indicar el tamaño de la pila de operandos del marco de activación del método correspondiente, y para indicar el cardinal del conjunto de variables locales del método correspondiente. Para hacer referencia a la pila de operandos, se escribe

.limit stack <***tamaño de la pila***>

Aquí podemos encontrarnos con un problema, que ocurre cuando existe la posibilidad de hacer optimizaciones en el código fuente (como reza la motivación de este proyecto), el tamaño de la pila podría variar. Una solución es poner más tamaño de pila del que se sepa con seguridad que se va a necesitar. Para hacer referencia al conjunto de variables locales se escribe

.limit locals <***número de variables locales***>

.end method – Indica la terminación de un método
.throws <***nombre de la clase***> - Indica que tipo de excepción puede ser lanzada por el método en el que se escribe esta directiva
.catch <***nombre de la clase***> ***from*** <***etiqueta1***> ***to*** <***etiqueta2***> ***using*** <***etiqueta3*** > - Esta directiva es la que reconstruye la tabla de excepciones en ***Jasmin***. Se tiene que declarar dentro de cada método que posea al menos un manejador de excepciones. El campo <***nombre de la clase***> indica el tipo de excepción que es susceptible de ser lanzada; los campos <***etiqueta1***> y <***etiqueta2***> indican el rango del código en el que se puede producir la excepción, el índice inicial y final, respectivamente; y el campo <***etiqueta3***> indica el índice destino donde se encuentra el manejador de la excepción.
.***line*** <***número de línea***> – Se usa para asignar un número entero a un fragmento de código. El conjunto de instrucciones de ***Jasmin*** etiquetado con un determinado número de línea se corresponderá con la línea concreta de código del programa fuente de Java. Se usa para depuración, ya que es útil saber en qué línea se ha producido un fallo
.var <***número de variable***> ***is*** <***nombre***> <***descriptor***> ***from*** <***etiqueta1***> ***to*** <***etiqueta2***> - Al igual que con la directiva que etiqueta una línea de código, esta directiva se usa para las opciones de depuración. En esta ocasión se etiqueta una variable local (la correspondiente al número de variable que se le indique, del conjunto de variables locales de cada método), con un nombre y un descriptor, y el rango en el que es visible.

4.2.2 INSTRUCCIONES

El conjunto de instrucciones de ***Jasmin*** es muy similar al bytecode, salvo pequeñas excepciones

La instrucción ***invokespecial*** se traduce como ***invokenonvirtual***

La instrucción ***invokeinterface*** se trata de manera diferente a como se le trata en el bytecode, ya que aparte del argumento que indica el método que queremos invocar de una determinada interfaz, han de ser especificados el número de parámetros que constan en el método, en concreto, el argumento será la suma de los parámetro de entrada y los de salida.

La manipulación de atributos también es diferente, ya que además del argumento que indica el nombre del atributo, después hay que especificar el tipo de ese atributo, mediante una notación mnemotécnica, que es la siguiente

Byte -> B
Char -> C
Double -> D
Float -> F
Int -> I
Long -> J
Short -> S

Boolean -> Z
Referencia a objeto -> L

Aunque en el último caso, las referencias a objetos, habría que escribir su notación mnemotécnica (L), seguida del nombre de la clase instanciada.

4.2.3 ETIQUETAS

Las etiquetas se usan en sustitución de las referencias simbólicas. En ***Jasmin*** consisten en una cadena de texto (que siempre debe empezar por una letra) seguido del carácter de dos puntos (‘***:***’). Su función es igual a la de los leguajes ensambladores ya que funcionan como destino de las instrucciones de transferencia de control. Además, son locales a cada método, es decir, se puede repetir una misma etiqueta en métodos distintos.

4.2.4 ESTRUCTURA DE UNA CLASE

Cuando se realice la escritura de una clase en ***Jasmin***, según todas las directivas que se han explicado, lo primero que debe figurar es información sobre la clase, es decir, al principio deben aparecer las directivas ***.class*** o ***.interface***, y ***.super***; y opcionalmente la directiva ***.implements*** en caso de implementar una interfaz.

Una vez declarada la cabecera del archivo, lo siguiente son las declaraciones de los atributos, es decir, de las directivas ***.field***; y por último las declaraciones de las cabeceras de los métodos, añadiendo la directiva de lanzamiento de excepciones en caso de la existencia de manejadores en ese método, y a continuación todas las instrucciones de ***Jasmin*** propias de ese método. En caso de que el método sea abstracto, irá indicada esa opción en el modificador de acceso de su signatura, y no contendrá ninguna instrucción.

4.2.5 EJEMPLO

Ahora vamos a ver un ejemplo completo en la sintaxis de Jasmin, más concretamente, el ejemplo antes descrito en la sección 2.8, la clase ***misc/students/test***. Al carecer de método ***main***, esta clase no se podría ejecutar directamente

```
.class  misc/students/test
.super java/lang/Object


.method  <init>()V
.limit stack 2
.limit locals 1
label1:
aload_0
```

```
label3:
invokenonvirtual java/lang/Object/<init>()V
label4:
return
.end method

.method  m(I)Lmisc/students/Tree;
.limit stack 12
.limit locals 2
label6:
iload_1
ifle label7
label8:
new misc/students/Tree
dup
aload_0
iload_1
iconst_1
isub
label11:
invokevirtual misc/students/test/m(I)Lmisc/students/Tree;
label12:
aload_0
iload_1
iconst_1
isub
label13:
invokevirtual misc/students/test/m(I)Lmisc/students/Tree;
label14:
iload_1
invokestatic misc/students/test/f(I)I
label15:
label16:
invokenonvirtual misc/students/Tree/<init>(Lmisc/students/Tree;Lmisc/students/Tree;I)V
label17:
areturn
label7:
aconst_null
areturn
.end method

.method static f(I)I
.limit stack 8
.limit locals 3
label18:
iconst_0
istore_1
iload_0
istore_2
label19:
iload_0
iconst_1
if_icmple label20
label21:
iload_1
new java/lang/Integer
dup
iload_0
label25:
invokenonvirtual java/lang/Integer/<init>(I)V
label26:
invokestatic misc/students/test/g(Ljava/lang/Integer;)I
label30:
iadd
istore_1
iload_0
iconst_2
```

```
label32:
idiv
istore_0
goto label19
label20:
iload_2
ifle label22
label23:
iload_1
new java/lang/Integer
dup
iload_2
label28:
invokenonvirtual java/lang/Integer/<init>(I)V
label29:
invokestatic misc/students/test/g(Ljava/lang/Integer;)I
label31:
imul
istore_1
iinc 2 -1
goto label20
label22:
iload_1
ireturn
.end method

.method static g(Ljava/lang/Integer;)I
.limit stack 4
.limit locals 1
label33:
aload_0
label35:
invokevirtual java/lang/Integer/intValue()I
label36:
iconst_1
iadd
ireturn
.end method
```

Figura 7. Clase **misc/students/test** en sintaxis de Jasmin

En este ejemplo no se pueden ver otras posibilidades representativas de ***Jasmin*** como la reconstrucción de la tabla de excepciones o la presencia de múltiples modificadores de acceso en los métodos, pero se puede comparar la sintaxis de ***Jasmin*** con la del bytecode para el ejemplo que hemos descrito en capítulos anteriores.

Capítulo 5

Traducción

En este capítulo serán abordados todos los aspectos de implementación del proyecto.

5.1 Lenguaje de implementación: Prolog

El lenguaje de programación con el que se ha desarrollado la implementación del compilador es el lenguaje Prolog. La razón para su uso es que la totalidad del sistema ***COSTA*** está implementado en este lenguaje. Como principal ventaja, se da que el tamaño del código va a ser mucho menor, y más compacto, frente a los lenguajes orientados a objetos, en los que se obtendrían un gran número de clases, para realizar la misma funcionalidad que se adquiere en Prolog con unos pocos módulos. Así mismo, la principal desventaja la encontramos en su ineficiencia, ya que la ejecución de Prolog está basada en la recolección de soluciones mediante backtracking, es decir, una estrategia de búsqueda de soluciones mediante fuerza bruta, lo que garantiza explorar todos los elementos posibles.

Prolog es un lenguaje de programación lógico, de propósito general y de quinta generación. Su utilidad está asociada con la Inteligencia Artificial y con la lingüística computacional. En su semántica está incluido un subconjunto puramente lógico, denominado “Prolog puro”, al que se le añaden unas extensiones de carácter extra-lógico.

Prolog tiene sus raíces en la lógica formal, y a diferencia de otros lenguajes de programación, Prolog es declarativo: el programa lógico es expresado en términos de relaciones, y su ejecución es disparada mediante consultas sobre estas relaciones. Las relaciones y las consultas se construyen usando el tipo de datos más simple de Prolog: el término. Las relaciones se definen mediante

cláusulas. Dada una consulta, el motor de Prolog intenta encontrar una resolución de la refutación de la consulta negada. Si la consulta negada puede ser refutada; se concluye que la consulta, con la instanciación encontrada aplicada, es una consecuencia lógica del programa. Esto hace que Prolog (y otros lenguajes de programación lógicos) sean particularmente útiles para aplicaciones tales como el manejo de bases de datos, el trato de operaciones matemáticas simbólicas, y el desarrollo de compiladores, entre otras.

Prolog fue creado a principios de los años 70 en la universidad de Marsella. En sus inicios, se trataba de un lenguaje de programación completamente interpretado, hasta que se desarrolló un compilador capaz de traducir los programas de Prolog a un lenguaje objeto interpretado por una máquina virtual llamada ***WAM*** (*Warren Abstract Machine*). Desde entonces, Prolog se considera un lenguaje semi-interpretado. Prolog fue uno de los primeros lenguajes de programación lógicos, y hoy en día aún se sigue considerando como el más popular de este tipo de lenguajes, existiendo un gran número de implementaciones, tanto libres como comerciales. Aunque inicialmente sus objetivos estaban centrados en el ámbito del procesamiento del lenguaje natural, el uso del lenguaje ha evolucionado ostensiblemente siendo ahora básico en otras áreas como la demostración de teoremas, sistemas expertos, juegos, sistemas automatizados de respuesta, ontologías y sistemas sofisticados de control; y los entornos de desarrollo modernos de Prolog dan soporte a la creación de interfaces gráficas de usuario, así como a las aplicaciones administrativas que se coordinan mediante redes de interconexión de computadores.

Al estar enmarcado dentro del paradigma de los lenguajes lógicos, su ejecución dista en gran parte de los lenguajes imperativos y de los orientados a objetos, ya que no existen conceptos altamente populares, como el de secuencia o el de instrucciones de control. Sus cláusulas se componen de antecedente y un consecuente. El antecedente está formado por una secuencia de condiciones denominada secuencia de objetivos. Su ejecución se basa en dos conceptos: el backtracking (vuelta atrás) y la unificación.

Con la unificación, cada objetivo determina un conjunto de cláusulas susceptibles de ser ejecutadas. Cada una de estas cláusulas se denomina punto de elección. Prolog selecciona el primer punto de elección y sigue ejecutando el programa hasta determinar si el objetivo es verdadero o falso.

El backtracking, o la vuelta atrás es una estrategia para encontrar soluciones a problemas que satisfacen restricciones. Cuando un objetivo de Prolog es falso, entra en juego la búsqueda de soluciones mediante esta técnica, que consiste en deshacer todo lo ejecutado anteriormente, situando el programa en el mismo estado en el que estaba antes de llegar al punto de elección. Entonces se toma el siguiente punto de elección que estaba pendiente y continua la ejecución desde ahí. Al final del programa todos los objetivos serán verdaderos o todos serán falsos.

5.1.1 ENTORNO DE DESARROLLO: SWI-PROLOG

A pesar de que se podía haber elegido cualquier intérprete de Prolog para el desarrollo del proyecto (en sus inicios se eligió Ciao Prolog), la decisión final fue la de desarrollarlo con SWI-Prolog. Este intérprete es de código abierto, y se usa comúnmente para la enseñanza y para las aplicaciones web semáticas. Tiene un conjunto de características muy rico en recursos, como librerías que extienden las limitaciones de la programación lógica, programación multi-hilo, unidades de pruebas y depurador gráfico, interfaces gráficas de usuario, y un servidor web, entre otras muchas. Es un intérprete multi-plataforma, que funciona en equipos Windows, Unix y Macintosh.

Uno de los objetivos de desarrollar el lenguaje en Prolog es utilizar ISO-Prolog, es decir, Prolog standard, para poder ejecutarlo en cualquier sistema. (Se podría ir más allá y utilizar únicamente Prolog puro; sin embargo, llevar a cabo esta idea alcanzaría unos niveles de dificultad mucho mayores al no estar incluidos los cortes en ese subconjunto)

5.2 Decompilación fichero .class a representación de reglas

Como se ha especificado en capítulos anteriores, el paso previo a la traducción realizada en este proyecto, es la decompilación del código de byte de Java a la representación intermedia basada en reglas.

Con el fin de complementar el entendimiento del proceso de compilación y su proceso inverso, se van a describir brevemente las dos principales etapas de esta decompilación.

- Se genera un grafo de control de flujo para cada método existente en el programa original en bytecode, usando técnicas básicas en el ámbito de compilación [*Aho et al. 1974; Aho et al. 1986*]. Algunas características más avanzadas, como pueden ser las invocaciones a métodos virtuales o la generación de excepciones son consideradas simplemente como nodos y aristas adicionales del grafo.
- Cada grafo de control de flujo se representa como un conjunto de procedimientos (compuestos de una o más reglas) usando una representación recursiva basada en reglas, donde los argumentos de cada instrucción del bytecode están representados explícitamente, y la pila de operandos se allana convirtiendo su contenido en variables locales adicionales.

El control de flujo de un programa es una información esencial para hacer razonamientos sobre su coste (que es el objetivo final de ***COSTA***), ya que permite razonar sobre todos los posibles caminos que pueden ser tomados durante la ejecución. También permite agrupar secuencias de instrucciones que siempre sean ejecutadas como un solo bloque. A diferencia de los programas escritos en alto nivel, escritos en lenguajes de programación estructurados como Java, los programas escritos en bytecode se caracterizan por sus flujos de control desestructurados.

5.3 Estructura de módulos

El proyecto se compone de 5 módulos básicos

BYTECODE_WRITER_UTILS

Es el módulo en el que están definidos predicados que no tienen relación con ninguna acción en particular, bien porque su objetivo es insignificante comparado con otros módulos, o bien porque son acciones que se repiten una y otra vez

concat/3 – Concatena o bien dos átomos; o bien una variable unificada y un átomo; o bien dos variables unificadas. Éste predicado ya existe en determinados entornos de desarrollo, como SWI-Prolog; sin embargo, ha sido necesaria su definición al no tratarse de ISO-Prolog.

concat2/3 – Realiza una función análoga al predicado anterior; sin embargo, en esta ocasión es especificada como argumento de entrada la cadena de caracteres final, y lo que se obtiene como salida es un fragmento de la cadena.

white_concat/2 – Concatena un átomo con el carácter ASCII número 32, que representa el espacio en blanco. Se utiliza para la traducción individual de cada instrucción.

make_label/2 – Concatena la cadena de caracteres ***"label"*** con el argumento que se especifica como entrada, generalmente será un número entero. Se usa para la generación de etiquetas necesaria para la creación de los archivos de ***Jasmin***.

filter_label/2 – Elimina el carácter ASCII imprimible "dos puntos" (***:***), de la etiqueta que se especifica como entrada. Se usa para el parcheo de manejadores de excepciones, ya que será necesario comparar etiquetas, y para ello es necesaria la eliminación de ese carácter.

concat_list_items/2 – Concatena todos los elementos de una lista formando una cadena de caracteres con todos ellos. Generalmente esos elementos serán, a su vez, cadena de caracteres. En la cadena de salida todos los elementos de salida figuran separados por un espacio. Se usa para la concatenación de todos los modificadores de acceso de una clase, método o atributo.

write_list_items/1 – Escribe en fichero todos los elementos de una lista, que suelen ser cadenas de caracteres. Se usa para imprimir las instrucciones ***tableswitch*** y ***lookupswitch***, ya que se asertan en la base de datos como listas de cadenas de caracteres, al estar definidas como una tabla.

count_number_classes/1 – Cuenta el número de clases que han sido decompiladas, es decir, traducidas a la representación intermedia para luego

ser traducidas otra vez al bytecode. Se usa como dato informativo en las tablas de tiempos

count_number_methods/1 – Análogo al predicado anterior, contando los métodos decompilados en vez de las clases. También se usa como método informativo en las tablas de tiempos.

class_access_resolution/2 – Indica cuales son los modificadores de acceso de una clase. Pueden ser public, final, interface y abstract. Estos atributos de las clases aparecen codificados tras la decompilación como un número entero. Para decodificarlo, se selecciona el modificador correspondiente haciendo una máscara con un determinado flag de un número en hexadecimal. Si al hacer la máscara obtenemos el valor lógico '1', ese modificador de acceso es añadido a la lista de salida.

field_access_resolution/2 – Hace la misma operación que el predicado anterior, con la excepción de que en esta ocasión son tratados los modificadores de acceso de los atributos, que pueden ser public, private, protected, static, final, volatile y transient.

method_access_resolution/2 – Hace la misma operación que los dos predicados anteriores, esta vez tratando los métodos. Los posibles modificadores de acceso son: public, private, protected, static, final, synchronized, native, abstract y strict.

BYTECODE_WRITER_RULES

Es el módulo encargado de la traducción de instrucciones a la sintaxis que acepta ***Jasmin***. Principalmente, consultará los hechos dinámicos ***bc/5*** asertados en la base de datos, y transformará el último parámetro de estos hechos, que se corresponde con la instrucción de bytecode adaptada a la representación de reglas. Habrá una correspondencia biyectiva instrucción a instrucción, en gran parte de casos, ambos formatos de instrucción son similares.

Además de esto, también contiene algunos predicados de menor importancia, que se dedican sobre todo a filtrar la información importante que el proceso de decompilación aserta en la base de datos; y predicados auxiliares.

transform/2 – Traduce expresiones de bytecode adaptadas de la representación intermedia basada en reglas de ***COSTA***, a instrucciones de ***Jasmin***. Existe una predicado ***transform*** por todas y cada una de las 256 instrucciones que conforman el bytecode. También genera las etiquetas, ya que éstas también están tratadas en el último argumento de los hechos dinámicos ***bc/5***. Como algunas de esas traducciones es más compleja que otras, será necesario el uso de varios predicados auxiliares.

resolve_type_reference/2 – Los tipos numéricos largos, los que aparecen en coma flotante y las referencias a String, aparecen en la representación de

reglas de manera diferente al resto de tipos primitivos. Este predicado filtra el valor de cada uno de estos tipos. Se usa en instrucciones de la familia ***ldc***.

resolve_field_reference/2 – Filtra de la información de la base de datos, los elementos importantes para la resolución de los atributos, es decir, el nombre de la clase y el nombre del atributo. Se usa en las instrucciones de acceso y manipulación de atributos.

resolve_method_reference/2 – Filtra de la información de la base de datos la información importante para la resolución de los métodos; esto es, el nombre de la clase y el nombre del método. Se usa en instrucciones de invocación a métodos.

resolve_method_interface_reference/2 – Se diferencia el caso de la instrucción ***invokeinterface*** por que su sintaxis es distinta en ***Jasmin*** de su sintaxis en el bytecode original, ya que recibe un número distinto de argumentos, el nombre de la interfaz, del método en cuestión, y el número de parámetros, que será la suma de los argumentos de entrada y de salida.

resolve_case_reference_lookupswitch/2 – Mención especial merecen los ***switches***, ya que hay que reconstruir una tabla de saltos considerándola como una sola instrucción. La instrucción ***lookupswitch*** guarda parejas <***indice, etiqueta***> en una lista, incluyendo también el salto por omisión, es decir, cuando el valor que entra en el ***switch*** no encaja con ninguno de los casos. Este predicado guarda una lista de cadenas de caracteres en la que cada pareja <***indice, etiqueta***> es un elemento de la lista.

resolve_case_reference_tableswitch/2 – La otra manera de codificar ***switches***, esta vez, para instrucciones ***tableswitch***, que difieren de los ***lookupswitch*** en que sus índices se pueden agrupar en un rango ordenado. En esta ocasión se procede de la misma manera que en el predicado anterior, guardando cadenas de caracteres en una lista, una por cada caso (aparte de la propia instrucción), para reconstruir la tabla de saltos. Sin embargo, ahora en vez de guardar por cada caso parejas <***indice, etiqueta***>, se guardan sólo las etiquetas. El único cambio es que ahora se debe especificar el índice inicial al lado de la instrucción, y al ser un rango ordenado, se entiende que se suma una unidad al índice inicial para indicar el índice de cada salto a cada etiqueta que vendrá especificada después. Lo que sí se tiene que indicar es el caso por omisión y la etiqueta por la que seguirá el flujo de ejecución. Este predicado crea esa lista de etiquetas.

resolve_class_name/2 – Consulta la información de la representación de reglas asociada a una clase y filtra el nombre de la clase, que es el argumento que devuelve como salida.

resolve_package_name1/2 – Al igual que el anterior, consulta la información asociada a una clase de la representación de reglas, pero esta vez devuelve el primer elemento de la ruta de la clase, es decir, el primer directorio del árbol de directorios de esa ruta. (Se utiliza para filtrar los ejemplos ***jolden***)

resolve_package_name2/2 – Análogo al anterior, pero esta vez filtra los dos primeros elementos de la ruta. (Se usa para seleccionar los ejemplos de las excepciones, ubicados en el directorio ***misc/studentsExceptions***).

resolve_limit_stack/2 – Este predicado resuelve el tamaño de pila que se le asignará a cada método, es decir, crea la directiva de ***Jasmin .limit stack***, con la única salvedad de, con el objetivo de hacer optimizaciones de código en el futuro, se le asigna a cada método el doble del tamaño de pila original de cada método, ya que el tamaño original se guarda en la representación de reglas. Este predicado consulta y filtra la información de esa representación y fija el nuevo tamaño creando la directiva.

resolve_local_variables/2 – Al igual que el predicado anterior, éste consulta la información adicional de cada método en la representación intermedia de reglas y filtra la relacionada con el tamaño del conjunto de variables locales. Una vez habiendo hecho ese filtrado, crea la directiva de ***Jasmin .limit locals***, pero esta vez no se modifica el tamaño de ese conjunto.

PRINT_FILE_JASMIN

Este módulo es el que se encarga de la impresión en fichero de la información traducida en la base de datos de Prolog. Para el tratamiento directo de ficheros se usan funciones de ISO-Prolog.

Los archivos se almacenarán en el directorio ***src/tmp*** y su nombre será el nombre de la clase decompilada y traducida. La extensión de los archivos traducidos será ***.j***, ya que es la que requiere ***Jasmin***. El nombre completo de la clase traducida, que es el que se indicará al traducir el archivo, es el nombre del paquete seguido del nombre de la clase. Los predicados que conforman este módulo son:

print_file/0 – El predicado general, consulta el nombre de la clase y crea un fichero con ese nombre.

make_header/1 – Crea la cabecera del fichero de ***Jasmin***, es decir, escribe en el fichero las directivas ***.class*** o ***.interface***, ***.super***; y ***.implements*** en el caso de que alguna interfaz sea implementada.

print_class/2 – Este predicado tiene dos flujos de ejecución distintos. Si al obtener los modificadores de acceso de la clase está entre ellos interface, quiere decir que se trata de una interfaz en vez de una clase, así que continúa por el flujo de ejecución en curso. Si esa condición no se cumple, entonces se trata de una clase. Este predicado es invocado por ***make_header/2*** y crea las directivas ***.class*** o ***.interface***.
print_implements/1 – Este predicado es llamado por ***make_header/2*** y su función es crear la directiva de ***Jasmin .implements***, en el caso de que la clase implemente alguna interfaz. El argumento que recibe es la lista de interfaces implementadas, un dato recogido al consultar la información de la

representación de reglas sobre la clase; si la lista es vacía, no se implementa ninguna interfaz.

print_file_aux/0 – Contiene el grueso de la impresión, hace llamadas a predicados auxiliares para escribir los atributos, las excepciones y otros metadatos, y crea las directivas de ***Jasmin .method*** y ***.end method***.

print_file_aux2/1 – Este predicado es invocado una vez por cada método de la clase, ya que el método es el argumento que recibe como parámetro. Consulta las instrucciones traducidas que contienen los hechos dinámicos ***bc/5*** y escribe en el fichero todas las instrucciones del método correspondiente, diferenciadnos si se trata de una sola instrucción o de una lista de instrucciones (el antes mencionado caso de los ***switches***).

print_stack_and_local_variables/1 – Escribe en fichero las directives ***.limit stack*** y ***.limit locals*** que crean los predicados ***resolve_limit_stack/2*** y ***resolve_local_variables/2*** del módulo ***bytecode_writer_rules***.

print_exception_table/1 – Crea la directiva ***.catch*** de ***Jasmin*** por cada método existente en la clase, que es el argumento que recibe el predicado. Consulta los hechos dinámicos ***handler/5*** de la base de datos, que es donde aparecen especificadas las excepciones y reconstruye la tabla de excepciones con la correspondiente directiva.

print_fields/0 – Crea las directivas ***.field*** de ***Jasmin***, tantas como atributos contenga la clase, y las escribe en fichero.

PRINT_TIME_TABLES

Es el módulo que se encarga de calcular los tiempos para los ejemplos que se han probado, aparte de proporcionar otra información.

Más concretamente, existirán dos ficheros que contienen tablas de tiempos, y los campos de las tablas son:

- Nombre del ejemplo
- Número de clases
- Número de métodos
- Número de reglas
- Tiempo fase decompilación
- Tiempo fase traducción y aserción en base de datos
- Tiempo fase impresión a fichero

Para llevar a cabo esta idea ha sido necesaria la utilización de otro módulo llamador, si bien antes era el ***decompiler***, ahora se ha introducido el módulo ***decompiler_jasmin***, en el mismo directorio ***src/driver***, cuya función es similar, exceptuando que en él se introducen más temporizadores, y ya se separa completamente las fases de traducción e impresión a fichero para tener un

desarrollo más modular y para calcular con más facilidad los tiempos especificados anteriormente.

print_time_tables/4 – Es el único predicado de éste módulo, ya que algunas funciones que solicita están definidas en otros módulos. En él tienen lugar dos posibles flujos de ejecución, ya que según la ubicación de los ejemplos, clasificamos éstos en dos posibles tablas.

WRITER

Se trata del módulo encargado de realizar el proceso de traducción desde las reglas de la representación intermedia.

El resultado es un conjunto de hechos Prolog, que es tratado por el resto de módulos. El algoritmo que sigue el módulo está explicado de manera extensa en los dos siguientes apartados.

Aparte de los predicados que se definen a continuación, consta de los hechos dinámicos ***bc/5***, ***handler/5***, ***bc_meter/1***, ***next_block/1***, ***top/1***, ***tag/1***, y ***goto/3***.

writer_rbr_to_class/0 – Predicado principal del módulo, que se sirve del resto para realizar el proceso de traducción. Tras un limpiado de la base de hechos, producidos en anteriores ejecuciones, recoge información sobre el nombre de la clase y métodos a procesar. Después realiza el proceso para todos los métodos disponibles, y por ultimo realiza un post-proceso consistente en la eliminación de etiquetas muertas y resolución de etiquetas en el contexto de los rangos de los manejadores de excepciones.

writer_rbr_to_class2/0 – Consiste en un bucle que realiza la traducción para cada uno de los métodos disponibles.

writer_rbr_to_class3/2 – Inicializa contadores y estructuras de datos necesarios para el proceso de traducción de cada uno de los métodos.

writer_rbr_to_bc/2 – Este predicado es el que analiza las reglas asertadas en la base de datos, y dependiendo de la estructura de éstas, las va tratando según corresponda. Se va ejecutando de manera recursiva para tratar todas las reglas e ir formando el código.

recovery/0 – En determinadas circunstancias, ***writer_rbr_to_bc/2*** no puede continuar ejecutándose porque no hay mas reglas en la pila de procesado. Este predicado comprueba si existe alguna etiqueta que todavía no ha sido tratada, y recupera la ejecución de ***writer_rbr_to_bc/2*** en caso afirmativo.

multiple_rule_writer/1 – Cuando varias ***rbr-rules*** tienen un identificador común hay que realizar un tratamiento especial, consistente en la preparación de la pila de procesado.

create_tag/3 – Predicado encargado de insertar etiquetas en la lista de instrucciones. Además comprueba si la instrucción predecesora es un salto a esa etiqueta, y en caso afirmativo elimina el salto.

remove_goto/1 – Elimina todos los saltos hacia una etiqueta pasada como parámetro.

remove_bc_goto_predecessor/1 – Comprueba si la instrucción introducida mas recientemente no es un salto hacia una etiqueta pasada como parámetro. En caso afirmativo elimina el salto y actualiza los contadores de instrucción.

process_current_rule/4 – Procesa un bloque de bytecodes contenidos en una ***rbr-rule***, y los introduce en la base de hechos Prolog como hechos dinámicos ***bc/5***.

process_bytecode/3 – Predicado que trata cada bytecode. Dependiendo de la clase realiza unas acciones determinadas, que sobretodo son actualizaciones en la pila de procesado, e inserciones de saltos.

insert_goto_if_necessary/3 – Debido a que las ***rbr-rules*** tienen información de flujo de control; pero pierden información estricta del orden de los bytecodes, cuando es necesario, hay que introducir hechos dinámicos del tipo ***goto/3*** para preservar el flujo del código resultante.

findMethods/2 – Evita la duplicidad en las llamadas a métodos.

makeTableSwitch/4 – Se encarga de insertar en la pila de procesado los bloques referentes a los distintos casos del ***switch***.

makeLookupSwitch/5 – Análogo al anterior, pero esta vez en las ocasiones en las que el ***switch*** es traducido con las instrucción ***lookupswitch***, ya que se tratan de diferente manera.

makeExceptionTable/2 – Dada una ***rbr-rule***, distingue el caso de la ejecución en el caso normal, y la ejecución en el caso de haber alguna excepción. Introduce los manejadores de excepción mediante el uso del predicado ***makeHandlers/2***.

makeHandlers/2 – Para cada bloque de ejecución perteneciente a un manejador de excepción, comprueba si se trata de una excepción definida por el programador mediante el uso de ***trueIfHandlerExists/1***, y en caso afirmativo inserta el manejador como un hecho Prolog ***handler/5***.

trueIfHandlerExists/1 – Comprueba si un manejador es definido por el programador o es inferido por ***COSTA***.

patchHandlers – Predicado perteneciente a la fase de post-proceso, que calcula una vez completado el bytecode del método, el rango en el que se puede dar una excepción para un determinado manejador.

***remove_dead_gotos* –** Predicado de la fase de post-proceso que elimina los saltos a etiquetas que no tienen código.

count_rbr/2 – Predicado que devuelve el numero de ***rbr-rules*** con un identificador determinado.

push_top/1 – Predicado que apila un identificador de regla en la pila de procesado.

load_top/1 – Predicado que desapila y devuelve la cima de la pila de procesado.

OTROS MÓDULOS

Para el completo funcionamiento del proyecto, se necesita enlazar nuestro conjunto de módulos con otras unidades existentes en ***COSTA***. Las más importantes son:

Decompiler – Es el modulo raíz de todo el proceso, ya que genera el grafo de control de flujo, invoca al proceso de decompilación e informa de posibles errores

Decompiler_jasmin – Es una copia del módulo raíz incluida por nosotros, que además de las funciones anteriores, invoca el proceso de compilación a bytecode, al de impresión en fichero y hace las mediciones de tiempos.

Tests – Contiene todos los benchmarks que se podrán probar para la decompilación, cualquier ejemplo que se quiera probar es necesario incluirlo en este módulo.

Utils – Contiene una serie de predicados que realizan funciones extra-lógicas, similares a las que proporcionan las librerías incluidas en varios entornos de desarrollo, pero que no forman parte del conjunto exclusivamente lógico de ISO-Prolog, se incluyen para resolver los posibles conflictos al interpretar el código en diferentes entornos de desarrollo.

Bytecode_reader – Es el conjunto de módulos que implementan el proceso de decompilación, es decir, el proceso inverso del que tiene lugar en este proyecto.

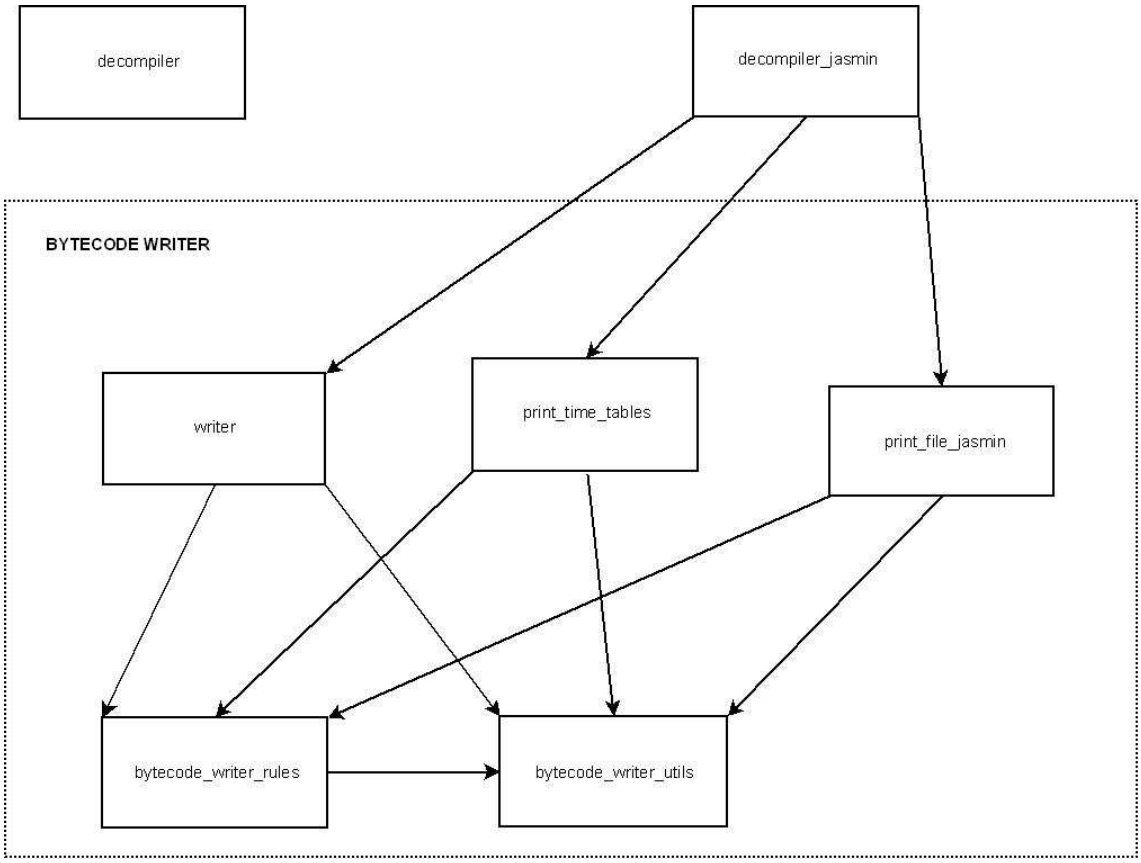

Figura 8. Estructura de módulos del compilador

5.4 Traducción: Primera versión, flujo de control sencillo

El proceso de traducción parte de hechos ***rbr-rule***, representados por hechos Prolog de la estructura ***rbr_rule (Regla Actual, Lista de bytecodes)***, y tiene como salida una serie de hechos Prolog con la estructura ***bc(Contador, Tipo, NombreClase, NombreMétodo, Bytecode)***, referidos al conjunto de bytecodes y etiquetas del programa, y una serie de hechos ***handler/5*** referidos a los manejadores de excepciones, cuya estructura y funcionamiento es explicada en el siguiente apartado. Después, a partir de estos hechos, se construye un archivo de texto que sea interpretable por ***Jasmin***, para obtener el bytecode ejecutable, equivalente al original.

El orden de procesado de las reglas ***rbr-rule***, viene dado por el uso de una pila en la que se van apilando y desafilando los identificadores de las reglas a tratar en cada momento. Así se puede soportar todas las estructuras de control de flujo que precisan instrucciones, como por ejemplo saltos condicionales anidados, ***switches***, o manejadores de excepciones.

El proceso de traducción tiene las siguientes fases, que son simplificadas por la complejidad y número de casos especiales a la hora de procesar las reglas:

1. Eliminación de hechos referentes a ejecuciones anteriores del predicado general.

2. Para cada uno de los métodos de la clase en proceso de traducción, hay que realizar los siguientes procesos:

    a. Obtenemos la primera ***rbr-rule*** del método actual, cuyo identificador viene dado por el nombre de la clase y método, la tratamos, obtenemos la siguiente regla y apilamos su identificador en la pila de procesado.
    b. Cada regla tiene entre otros componentes: Un identificador, una posible guarda con condición, una lista de bytecodes, y un enlace con la siguiente ***rbr-rule***, que da forma al grafo del flujo de ejecución de la representación intermedia. Partiendo de la regla actual, hay que realizar las siguientes acciones:

        i. Obtener el conjunto de reglas actuales desapilando de la pila de procesado. Pueden ser varias, porque varias reglas pueden tener el mismo identificador. Éste es el caso que se produce cuando el flujo de ejecución depende de cierta condición establecida por una guarda.
        ii. Apilar el identificador de la siguiente o siguientes reglas en la pila de procesado. El número de identificadores a apilar depende del número de reglas que tengan el mismo identificador.
        iii. Inserción de una etiqueta en el código, mediante un hecho ***bc/5***. Es necesario introducirla porque otros bytecodes pueden necesitar realizar saltos a esa parte del código.
        iv. Tratamiento de la lista de bytecodes de la regla. El tratamiento de cada tipo de bytecode es independiente y tiene fases distintas, pero todos tienen unas fases comunes:

            1. Traducción del bytecode al formato aceptado por ***Jasmin***, mediante el uso del modulo ***bytecode_writer_rules***, y obtención de la totalidad de los parámetros del bytecode.
            2. En el caso de bytecodes referidos a instrucciones que conllevan un salto, se buscan las etiquetas de estos, y se actualiza la pila de procesado, apilando los identificadores de estos.
            3. Inserción del hecho ***bc/5*** correspondiente.

3. Eliminación de saltos a etiquetas que no tienen código asociado.
4. Obtención del rango de los manejadores de excepciones, lo cual es explicado con detalle en el siguiente apartado.

5.5 Traducción: Segunda versión, inclusión de excepciones

La segunda versión del traductor, contiene el tratamiento de las excepciones definidas por el programador.

COSTA en su traducción a la representación intermedia, pierde la definición de los manejadores, aunque conserva las rutinas del tratamiento de excepción. El siguiente ejemplo muestra como trata ***COSTA*** las excepciones:

```
Try
{
        c=a/b;
        args[1] = "";
}
        catch (Exception e)
        {
            c=1;
        }
```

Como se puede observar en el código java, hay definido un manejador para el bloque de instrucciones delimitado por el ***try***, perteneciente a la superclase ***Exception***, con lo que cualquier excepción producida en ese contexto conllevaría la ejecución del bloque ***catch***.

Con la utilidad ***Javap*** podemos obtener la tabla de excepciones para ese código:

Tabla de excepciones			
Desde	Hasta	Dirección rutina	Tipo
6	15	25	***Java/lang/Exception***

En la tabla de excepciones, solo hay una entrada, que engloba todas las instrucciones que hay en el bloque ***try (6-15)***. ***COSTA*** no sigue este formato de tablas de excepciones, ya que las optimizaciones de código que realiza hacen que la tabla de excepciones pueda ser incoherente con el código mejorado. En lugar de esto, las ***rbr-rules*** mediante el uso de guardas plantean un flujo de ejecución distinto para el caso en el que no ocurre ninguna excepción y el caso en el que una instrucción única la produzca.

En este ejemplo para la primera instrucción se producen las siguientes reglas

```
9:=
    assr(int_nonzero)
    call(block, 3)


9:=
    assr(int_zero)
    bytecode(-3, ie_throw(java/lang/ArithmeticException)
    call(block, 2)
```

La guarda ***assr*** comprueba si el divisor de la operación es igual a 0. En caso negativo sigue la ejecución normal yendo al bloque 3, y en el caso positivo va al bloque 2, donde se encuentra el código del manejador.

Como se puede observar en la segunda regla el tipo de excepción es del tipo ***ArithmeticException***, perdiéndose también la superclase usada por el usuario. Por tanto, ***COSTA*** generaría las mismas reglas para códigos java diferente.

La solución elegida para solucionar el problema es la de incluir un capturador específico para cada instrucción dentro del bloque ***try*** que puede generar una excepción. Para el ejemplo expuesto la tabla de excepciones que generaría el compilador sería:

Tabla de Excepciones			
Desde	Hasta	Dirección	Tipo
9	10	25	División por cero
12	13	25	Error de rango

Se puede observar que hay un manejador individual para cada instrucción, y este manejador es del tipo específico que puede ocasionar la excepción. Si hubiera varios tipos de excepciones posibles, habría una entrada por cada uno.

El proceso de traducción soportando el uso de excepciones no supone ninguna modificación respecto al que no las soporta, salvo en el hecho de introducir los hechos ***handler*** correspondientes a los manejadores de la tabla de excepciones. Esto se hace en la fase 2.b del apartado anterior, y simplemente es una comprobación de si el flujo para el caso de haber una excepción lleva a reglas con bytecodes definidos por el programador, y no bytecodes introducidos por ***COSTA***.

Esto sería suficiente de no ser por un problema con ***Jasmin***, que impide usar rangos de una sola instrucción. Por eso es necesario dejar el campo *hasta* de las entradas por rellenar, y posteriormente en la fase 4 del proceso de traducción, vista en el apartado anterior, realizar un parcheo, que consiste en la búsqueda de la etiqueta inmediatamente posterior a la instrucción que genera la excepción.

5.6 Impresión a formato Jasmin

Una vez ha terminado el algoritmo, quedan asertados en la base de datos varios predicados dinámicos. Concretamente, los que contienen la información que queremos imprimir en fichero son los hechos ***bc/5*** y ***handler/5***. La impresión en fichero es únicamente una llamada al predicado principal del módulo ***print_file_jasmin***, es decir, una llamada al predicado ***print_file/0***. Esta llamada se hace desde el módulo ***decompiler_jasmin*** y tiene lugar una vez ha acabado el algoritmo anterior. Se va reconstruyendo fase por fase el fichero de ***Jasmin*** tal y como se ha especificado en el capítulo anterior, sacando la información de la base de datos, en muchos casos mediante la técnica de bucles al fallo. Al terminar, se puede localizar el archivo ***.j*** en el directorio ***/src/tmp***, de nombre el nombre de la clase analizada.

5.7 Creación de fichero .class

Una vez obtenido el fichero ***.j*** lo único que resta por hacer es convertirlo a formato ***.class*** para que posteriormente pueda ser ejecutado por el intérprete de la máquina virtual de Java.

Para probar ejemplos simples, que consten de una única clase, es suficiente con pasar el archivo ***.j*** a la aplicación ***Jasmin***, como se explicó en el capítulo anterior, y el resultado es un archivo ***.class*** que ya se podrá ejecutar desde línea de comandos y ver su resultado. En el caso de que los ejemplos consten de muchas clases, iremos pasando cada clase traducida por ***Jasmin***, hasta completar todo el programa. La primera clase que ejecutaremos será la principal, la que contenga el método ***main***, y después, una manera interesante de probar que el compilador va funcionando, es ir probando a ejecutar el programa después de cada traducción de una clase, ya que desde la línea de comandos el verificador de la máquina virtual de Java irá informando de la clase que falta por traducir, y cuando ya haya pedido todas las clases, se podrán visualizar los resultados.

Capítulo 6

Tablas de tiempos

Para terminar, hemos hecho análisis en tiempo en cada una de las fases de la compilación y de la compilación inversa. Los tiempos que aquí figuran son una simple muestra que hemos obtenido de los ejemplos que se localizan en ***examples/jolden***, para ejemplos grandes con varias clases, y también los ejemplos ***examples/studentsExceptions***, que son sobre los que hemos hecho el estudio incluyendo las excepciones.

6.1 Ejemplos jolden

6.1.1 VORONOI

Este programa genera un conjunto aleatorio de puntos y realiza un diagrama de Voronoi para esos puntos.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
Voronoi	1	5	479	109 ms	140 ms	0 ms
Vertex	1	17	689	218 ms	390 ms	0 ms
Vec2	1	15	316	94 ms	94 ms	0 ms
MyDouble	1	2	23	15 ms	0 ms	0 ms
Edge	1	31	1766	625 ms	1250 ms	0 ms
EdgePair	1	3	30	31 ms	16 ms	0 ms

6.1.2 BISORT

Realiza una implementación del algoritmo paralelo óptimo “Bitonic Sort”, utilizado para la ordenación de computadores de memoria compartida. El ejemplo ordena N números, donde N es una potencia de dos. Si el usuario proporciona un valor de entrada que no es potencia de dos, entonces se usará la potencia de dos más cercana que quede por debajo del valor de entrada.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
BiSort	1	5	230	47 ms	47 ms	0 ms
Value	1	10	407	109 ms	172 ms	0 ms

6.1.3 BH

Realiza una implementación del algoritmo de simulación Barnes-Hut

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
BH	1	7	583	172 ms	187 ms	0 ms
Tree	1	8	815	250 ms	500 ms	0 ms
Node	1	1	136	47 ms	31 ms	0 ms
Body	1	17	605	187 ms	281 ms	0 ms
MathVector	1	21	641	156 ms	328 ms	0 ms
Cell	1	6	239	47 ms	78 ms	0 ms
Body$1Enumerate	1	3	44	16 ms	15 ms	0 ms
Body$2Enumerate	1	3	44	16 ms	0 ms	16 ms
Node$HG	1	1	40	0 ms	15 ms	0 ms

6.1.4 HEALTH

Este programa realiza una simulación del sistema de sanidad colombino, y ejecuta esta simulación para un tiempo específico.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
Health	1	5	657	188 ms	219 ms	15 ms
Village	1	9	453	93 ms	188 ms	0 ms
Hospital	1	5	389	94 ms	125 ms	0 ms
List	1	4	129	31 ms	31 ms	0 ms
List$ListEnumerator	1	3	50	16 ms	0 ms	0 ms
Patient	1	1	22	16 ms	0 ms	0 ms
List$ListNode	1	1	19	0 ms	0 ms	0 ms
Results	1	1	10	0 ms	0 ms	0 ms

6.1.5 TSP

Una implementación en Java del famoso problema del viajante de comercio.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
TSP	1	5	393	94 ms	109 ms	0 ms
Tree	1	11	808	281 ms	657 ms	0 ms

6.1.6 PERIMETER

Este programa calcula el perímetro total de una región en una imagen binaria representada mediante un QuadTree. Más concretamente, el algoritmo crea una imagen, cuenta el número de hojas del QuadTree, y luego calcula el perímetro de la imagen usando un algoritmo-samet.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
Perimeter	1	5	434	110 ms	140 ms	16 ms
QuadTreeNode	1	16	432	93 ms	172 ms	0 ms
WhiteNode	1	3	16	15 ms	0 ms	0 ms
BlackNode	1	3	141	31 ms	32 ms	0 ms
GreyNode	1	3	92	16 ms	31 ms	0 ms
Quadrant	1	4	28	16 ms	0 ms	0 ms
NorthWest	1	4	50	0 ms	0 ms	0 ms
NorthEast	1	4	50	0 ms	0 ms	0 ms
SouthWest	1	4	50	15 ms	16 ms	0 ms
SouthEast	1	4	50	15 ms	16 ms	0 ms

6.1.7 EM3D

Este programa modela la propagación de las ondas electromagnéticas a través de objetos en tres dimensiones. El cálculo se realiza en un grafo bipartito irregular cuyos nodos representan los valores de los campos eléctrico y magnético.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
Em3d	1	5	550	156 ms	187 ms	0 ms
BiGraph	1	4	598	172 ms	172 ms	0 ms
Node	1	10	336	94 ms	94 ms	0 ms
Node$1Enumerate	1	3	44	16 ms	15 ms	0 ms

6.1.8 MST

Este programa calcula el árbol de recubrimiento mínimo de un grafo, usando el algoritmo de Bentley.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
MST	1	8	712	203 ms	281 ms	15 ms
Graph	1	7	189	63 ms	63 ms	0 ms
Vertex	1	6	63	15 ms	16 ms	0 ms
Hashtable	1	5	214	62 ms	47 ms	0 ms
HashEntry	1	5	47	15 ms	16 ms	0 ms
BlueReturn	1	5	38	16 ms	15 ms	0 ms

6.1.9 POWER

Este programa implementa un algoritmo de programación paralela para calcular la potencia óptima descentralizada. Se fija el número de clientes a 10000. Se crea una estructura de datos que contiene la raíz (la estación de potencia). Existen 10 alimentadores principales desde la raíz y cada alimentador salta a 20 nodos laterales. Cada nodo lateral es la cabeza de una línea de 5 nodos de salto, y cada salto tiene 10 clientes. Pretende maximizar el consumo de potencia para una comunidad.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
Power	1	5	321	78 ms	78 ms	0 ms
Root	1	7	755	203 ms	437 ms	0 ms
Demand	1	14	600	187 ms	453 ms	0 ms
Lateral	1	2	270	94 ms	125 ms	0 ms
Branch	1	2	308	109 ms	125 ms	0 ms
Leaf	1	2	57	15 ms	16 ms	0 ms

6.1.10 TREEADD

Este programa realiza un recorrido en profundidad de un árbol binario de manera recursiva, y suma el valor de cada elemento. Por defecto, se inicializan todos los elementos a ‘1’.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
TreeAdd	1	5	386	94 ms	78 ms	0 ms
TreeNode	1	7	188	32 ms	47 ms	0 ms

6.1.11 OBSERVACIONES

Una vez analizado el tiempo del decompilador y de nuestro compilador, y habiendo visto el tiempo concreto de cada fase del proceso, encontramos algunos puntos a destacar:

El tiempo que transcurre en la fase de la impresión de los hechos dinámicos de la base de datos a fichero, siempre es despreciable, ya que en la mayor parte de las mediciones es de 0 milisegundos, siendo en algunas ocasiones de 15 o 16 milisegundos, que puede ser producido al buscar y abrir el fichero, porque siempre coincide con la primera clase que se analiza.

Por lo general, el tiempo que tarda el algoritmo creado en este proyecto, cuya finalidad es la aserción de diferentes hechos dinámicos en la base de datos, es aproximadamente el doble que el proceso inverso, para tamaños de datos razonablemente grandes, ya que para tamaños de datos más pequeños andan parejos los tiempos de ejecución. De cualquier manera, ambos costes de tiempo están en el mismo orden de magnitud.

6.2 Ejemplos studentsExceptions

El segundo diseño del traductor desarrollado en este proyecto, consta de la capacidad de detectar excepciones y de reconstruir la tabla de excepciones. Los ejemplos que se han probado con la inclusión de excepciones se encuentran localizados en el directorio ***examples/misc/studentsExceptions***. Sobre ellos también se podrán hacer optimizaciones, ya que al poder detectar de antemano si un objeto es una referencia a ***null***, se podrán eliminar del bytecode generado los manejadores que hagan referencia a la excepción ***NullPointerException***.

En esta ocasión, los ejemplos probados recogen un flujo de control muy sencillo, que solo está compuesto de unas pocas instrucciones y la detección de distintos tipos de excepciones. Los tiempos obtenidos son los que se muestran a continuación.

Nombre de la clase	*Número de clases*	*Número de métodos*	*Número de reglas*	*Tiempo fase decompilación*	*Tiempo fase traducción*	*Tiempo fase impresión*
exc1	1	2	17	0 ms	16 ms	0 ms
exc2	1	2	18	0 ms	0 ms	0 ms
exc3	1	2	18	0 ms	0 ms	16 ms
exc4	1	2	24	0 ms	0 ms	0 ms
exc5	1	2	24	0 ms	0 ms	0 ms
exc6	1	2	27	0 ms	0 ms	0 ms
excBranch	1	4	121	15 ms	32 ms	0 ms

Los ejemplos sobre los que se ha hecho análisis de tiempos no son lo suficientemente grandes como para sacar conclusiones, ya que constan de unos pocos métodos, y por lo general, los tiempos son despreciables. Sólo del último ejemplo de la tabla (***misc/studentsExceptions/excBranch***), se puede deducir que el coste en tiempo de programas con excepciones es similar al de

los programas sin ellas, es decir, aproximadamente el doble, así que también están en el mismo orden de magnitud.

Algunos de los ejemplos analizados en el apartado anterior también contenían excepciones, como por ejemplo la clase ***jolden/bh/MathVector***, de la que podemos sacar la misma conclusión.

Capítulo 7

Optimizaciones

Para finalizar, como se ha mencionado antes, la razón principal por la que tienen lugar las compilaciones entre la representación de reglas y el bytecode, aparte de la realización de los análisis de consumo de recursos y terminación sobre las reglas, es hacer optimizaciones sobre esta representación para finalmente obtener un bytecode más compacto y eficiente.

A pesar de que estas optimizaciones están fuera del ámbito de este proyecto, merece la pena mencionar las más importantes.

- Una de las opciones de los análisis de ***COSTA*** es la de detectar si una referencia a un objeto es el objeto nulo, es decir, es una referencia a ***null***. Si se puede predecir esta información por adelantado se podrían realizar dos mejoras inmediatas.
    - Si en el cuerpo de un bucle hay una comparación de una referencia a un objeto nulo, con la información que se puede obtener por adelantado, se podría suprimir esa comparación, siendo similar en las instrucciones de transferencia de control. Hemos comprobado experimentalmente que en un programa simple, con la inclusión de un bucle en las condiciones antes especificadas, para un número de iteraciones de un millón, la ganancia en tiempo con la supresión de la rama derivada de la condición eliminada es de aproximadamente 15 milisegundos (sin apenas instrucciones en el cuerpo del bucle).
    - La otra mejora inmediata hace referencia a las excepciones. Si se sabe de antemano si un objeto es el objeto nulo, es posible ahorrarse la captura de la excepción ***NullPointerException***; sin embargo, la ganancia en tiempo es despreciable en esta ocasión.
- Por último, otra de las opciones de ***COSTA*** nos permite predecir si existe una sola instancia de un método concreto, por lo que se podrían sustituir determinadas instrucciones de bytecode ***invokevirtual*** por

invokespecial. De esta última mejora no existen resultados experimentales.

Apéndice: Limitación con los números reales

Este proyecto tiene una limitación, y es que en programas en los cuáles aparezcan números reales, los resultados al ejecutar el programa origen son distintos a los que se obtienen al ejecutar el programa, una vez éste ha sido decompilado, vuelto a compilar y ensamblado en ***Jasmin***; es decir, se obtiene un resultado distinto cuando al programa se le aplican todas las transformaciones vistas en este proyecto.

Ésto es debido a la no completitud del proceso anterior, es decir, del decompilador, el cuál no traduce los valores float y double del bytecode original, sin embargo, el proyecto está desarrollado de manera que funcione correctamente una vez se haya solucionado ese problema.

La traducción de los valores float y double puede ocasionar un mínimo incremento de tiempo en la fase de decompilación; sin embargo, no modificará los órdenes de magnitud especificados anteriormente.

Lista de palabras clave para su búsqueda bibliográfica

- Compilador
- Prolog
- COSTA
- Jasmin
- Bytecode
- Java
- Análisis
- Optimización
- Regla
- Excepción

Bibliografía

- The Java Virtual Machine Specification by / Tim Lindholm, Frank Yellin, Second edition.
- The art of Prolog: advanced programming techniques by / Leon Sterling, Ehud Shapiro, with a foreword by David H. D. Warren. Cambridge; London : MIT Press 2001, 2nd edition.
- Compiladores: principios, técnicas y herramientas by / Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Addison-Wesley Iberoamerican 1990, First edition.
- Cost Analysis of Objected-Orient Bytecode Programs by / Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla and Damiano Zanardini.
- COSTA Home page http://costa.ls.fi.upm.es
- Jasmin Home Page http://jasmin.sourceforge.net
- Wikipedia (Español) http://es.wikipedia.org
- Wikipedia (English) http://en.wikipedia.org